Neural Computing and Applications
https://doi.org/10.1007/s00521-018-3761-1

REVIEW

A survey of FPGA-based accelerators for convolutional neural networks

Sparsh Mittal[1]

Received: 11 January 2018 / Accepted: 28 September 2018

Abstract
Deep convolutional neural networks (CNNs) have recently shown very high accuracy in a wide range of cognitive tasks, and due to this, they have received significant interest from the researchers. Given the high computational demands of CNNs, custom hardware accelerators are vital for boosting their performance. The high energy efficiency, computing capabilities and reconfigurability of FPGA make it a promising platform for hardware acceleration of CNNs. In this paper, we present a survey of techniques for implementing and optimizing CNN algorithms on FPGA. We organize the works in several categories to bring out their similarities and differences. This paper is expected to be useful for researchers in the area of artificial intelligence, hardware architecture and system design.

Keywords Deep learning · Neural network (NN) · Convolutional NN (CNN) · Binarized NN · Hardware architecture for machine learning · FPGA · Reconfigurable computing · Parallelization · Low power

1 Introduction

In recent years, CNNs have been successfully applied for a wide range of cognitive challenges such as digit recognition, image recognition and natural language processing. As CNNs continue to be applied for solving even more complex problems, their computational and storage demands are increasing enormously. Conventionally, CNNs have been executed on CPUs and GPUs; however, their low throughput and/or energy efficiency presents a bottleneck in their use. While ASIC can allow full customization and small form factor, it leads to complex design cycle, high initial investments and lack of reconfigurability which is especially crucial given the rapid improvements in CNN architectures.

Due to their several attractive features, FPGAs present as promising platforms for HW[1] acceleration of CNNs [1]. Generally, FPGAs provide higher energy efficiency than both GPUs and CPUs and higher performance than CPUs [2]. Further, unlike the case of application-specific instruction processor, the reconfigurability of FPGAs allows quick exploration of vast design space of CNN configurations/parameters. Historically, the large design time and need of HW expertise had restricted the use of FPGA; however, the HLS tools promise to address these challenges by facilitating automatic compilation from high-level program to low-level RTL specifications [3]. Also, by virtue of supporting customizable data types, FPGAs improve resource utilization efficiency.

Use of FPGA for accelerating CNNs, however, also presents challenges. For example, AlexNet has more than 60 M model parameters and storing them in 32b FP format requires $\sim$ 250 MB storage space [4]. This far exceeds the on-chip memory capacity of FPGAs, and transferring these values to/from off-chip memory leads to performance and energy overheads. Other networks such as GoogLeNet and VGG have even higher memory requirement. Further, due

✉ Sparsh Mittal
sparsh@iith.ac.in

[1] Department of Computer Science and Engineering, Indian Institute of Technology, Hyderabad, India

[1] Following acronyms are used frequently in this paper: bandwidth (BW), batch normalization (B-NORM), binarized CNN (BNN), block RAM (BRAM), convolution (CONV), digital signal processing units (DSPs), directed acyclic graph (DAG), design space exploration (DSE), fast Fourier transform (FFT), feature map (fmap), fixed point (FxP), floating point (FP), frequency-domain CONV (FDC), fully connected (FC), hardware (HW), high-level synthesis (HLS), inverse FFT (IFFT), local response normalization (LRN), lookup tables (LUTs), matrix multiplication (MM), matrix–vector multiplication (MVM), multiply–add–accumulate (MAC), processing engine/unit (PE/PU), register transfer level (RTL), single instruction multiple data (SIMD).

Published online: 06 October 2018

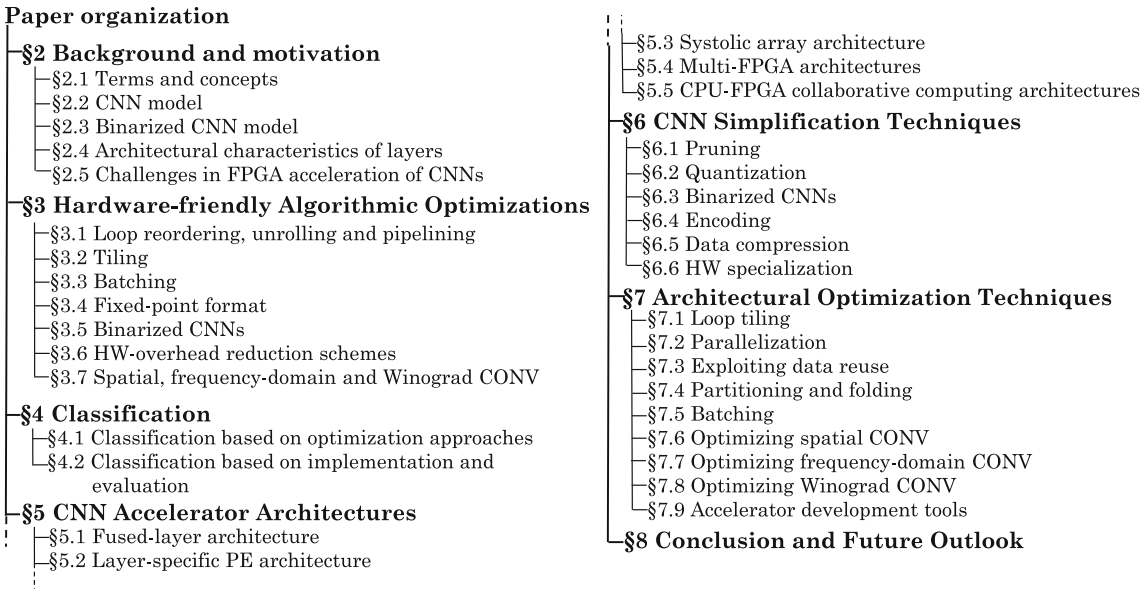


Fig. 1 Organization of the paper

to the unique architecture of FPGAs, accelerating quantized/binarized/pruned CNNs on FPGAs requires careful optimization.

Contributions In this paper, we present a survey of techniques for designing FPGA-based accelerators for CNNs. Figure 1 shows the overview of this paper. Section 2 presents a background on FPGAs and CNNs and identifies the challenges in FPGA-based acceleration of CNNs. Section 3 introduces commonly used optimization techniques, and Sect. 4 classifies the works based on key parameters. Section 5 discusses architectures of several FPGA-based CNN accelerators. Section 6 reviews techniques for simplifying CNN models which helps in reducing HW overhead of FPGA accelerators. Section 7 discusses techniques for optimizing throughput and energy efficiency of FPGA-based CNN accelerators. In these sections, we summarize a work under a single category, even though many of these works span across the categories. Section 8 closes the paper with an outlook on promising future directions and challenges.

Scope For sake of concise presentation, we limit the scope of this paper as follows. We include works which propose architectural improvements for CNNs for implementation on FPGA, and not those who use FPGA merely for quantitative comparison. Instead of discussing mathematical details or numerical results, we focus on the key ideas of each work to gain insights. This paper is expected to be useful for chip-designers and researchers in the area of machine learning and processor architecture.

2 Background and motivation

In this section, we first introduce some concepts (Sect. 2.1) which will be useful throughout this article. We then provide a background on CNNs (Sect. 2.2) and BNNs (Sect. 2.3) and discuss the architectural attributes of different layers (Sect. 2.4). Finally, we discuss the opportunities and obstacles in use of FPGA for implementing CNNs (Sect. 2.5).

2.1 Terms and concepts

Roofline model The roofline model relates system performance to computational performance and off-chip memory traffic. The execution of an algorithm may be bound by either computation or memory access, and hence, the maximum attainable performance is the minimum of (1) peak computational performance and (2) peak performance supported by the memory for a given compute-to-communication ratio. This is illustrated in Fig. 2.

Dataflow model In dataflow execution paradigm, an algorithm's functionality is divided into parallel threads (termed as "actors") and the data flows between them. This dataflow graph is statically mapped to the FPGA by using a library of compute entities which implement actors.

Systolic array The systolic array design uses deeply pipelined group of PEs. It achieves high clock frequency and low global data communication by virtue of using local communication and regular layout. This makes systolic array suitable for FPGA implementation.

Double buffering This scheme uses two buffers and allows the data of one buffer to be used while the data are

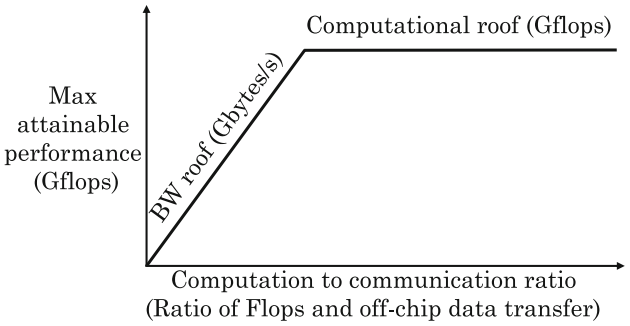


Fig. 2 Illustration of roofline model

filled in the next buffer. In the next phase, the roles of buffers are reversed, and thus, the buffers are said to work in "ping–pong" manner. This scheme allows hiding the data transfer latency.

FPGA architecture FPGA provides four types of resources: DSPs, LUTs, flip–flops and BRAMs. A DSP is an ALU which can perform logic/add/subtract/multiply operations. A DSP slice can implement a MAC operation. In Xilinx FPGAs, the DSP slice is also referred to as DSP48 since it can perform a maximum of 48-bit addition. An LUT can implement a combinatorial logic operation, whereas BRAMs are used for on-chip storage of data. FPGAs generally work at low frequencies (e.g., few hundreds of MHz) and depend on massive parallelism for achieving high throughput.

CPU–FPGA shared memory architecture In CPU–FPGA shared memory system, FPGA can access shared memory for performing reads/writes [5]. The CPU program can access whole memory, whereas FPGA can access only a part of it. For transferring data between the private memory of CPU thread and the shared memory, a "memory relay engine" is used which works as a CPU thread. The FPGA can also be seen as a thread which works on same data as the CPU. FPGA and CPU can simultaneously operate on different parts of shared data using a synchronization scheme.

2.2 CNN model

CNNs are composed of multiple layers, as we discuss below.

CONV layer The CONV layer works by sliding a large number of small filters across an image to extract meaningful features. Figure 3a depicts the data structures in a CONV layer, and Fig. 3b shows its pseudo-code (the bias is ignored). The inputs to CONV layer are N fmaps. Every fmap is convolved by a shifting window with a $K \times K$ kernel, which produces one pixel in one output

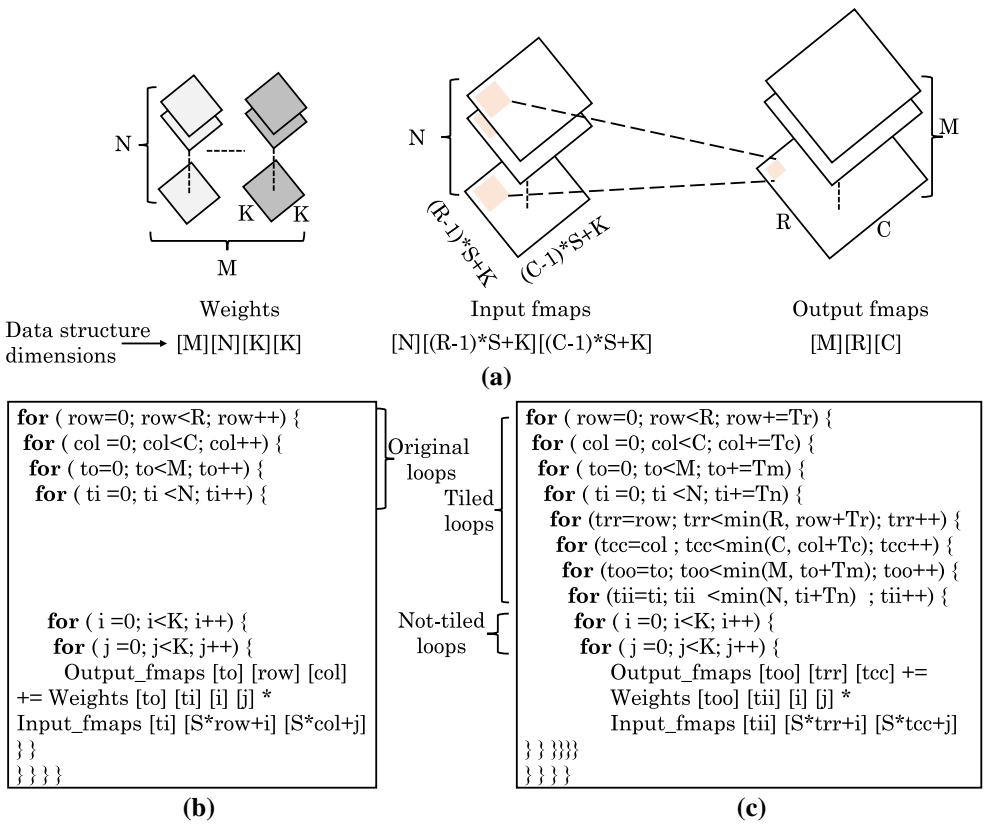


Fig. 3 **a** Illustration of CONV operation. Pseudo-code of CONV layer **b** without tiling and **c** with tiling (of selected loops)

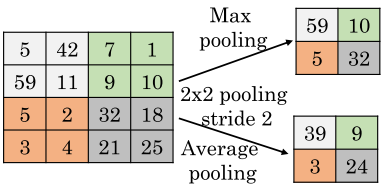


Fig. 4 Examples of max and average pooling

fmap. The shifting window has a stride of S which is generally smaller than K. The M output fmaps are taken as the input fmaps for the next CONV layer.

Pooling and nonlinear layers The pooling layers realize spatial invariance by subsampling adjacent pixels. Figure 4 shows commonly used pooling schemes, viz. average pooling and max pooling. Nonlinear layers emulate the activation of biological neuron by using a nonlinear function on each pixel of the fmap.

Fully connected layer As the name implies, neurons in FC layer have connections to all activations in the previous layer. FC layer performs classification on the features extracted by the CONV layers.

2.3 Binarized CNN model

Since CNNs can tolerate errors and can be retrained, recent work has proposed binarized CNNs which use binary weights and activations [6]. Figure 5 shows the comparison of the layers and data types in a CNN and a BNN [3]. In BNN [6], pooling happens early and B-NORM is performed before binarization for reducing information loss. Note that B-NORM is a strategy for adjusting the deviation of the biased data by performing shifting and scaling operations. Without using B-NORM, the average output value and activation of a BNN are unbalanced, which leads to poor accuracy. Use of B-NORM ensures that the average value of internal variable in the BNN can be nearly zero, which improves the accuracy.

2.4 Architectural characteristics of layers

For directing the HW acceleration efforts, we now discuss the characteristics of different layers of CNNs. CONV and FC layers are compute and BW intensive, respectively [7]. For example, in AlexNet network, CONV and FC layers have less than 5% and more than 95% of total weights, but they account for $\sim$ 93% and $\sim$ 7% of total computations, respectively [7]. Pooling and nonlinear layers do not require large amount of resources. FC layers are responsible for only 0.11% of the execution time of AlexNet [8]. This factor has decreased further in recent CNNs, e.g., GoogLeNet has 57 CONV layers and only one FC layer [9].

While nonlinear and FC layers have no data reuse, pooling layer has low data reuse. By comparison, CONV layer shows high data reuse. For example, in AlexNet, every weight of every kernel is used in all pixels of the corresponding output fmaps, which means that each data item is reused 3025 (= 55 $\times$ 55) times [10].

2.5 Challenges in FPGA acceleration of CNNs

Design of FPGA-based CNN accelerators poses several challenges, as we show below.

Trade-off between RTL and HLS approaches The HLS-based design approach allows quick development through high-level abstractions. As an example, Fig. 6 shows the generic design flow on using the OpenCL-based approach [4], which may also provide features such as automated pipelining and partitioning of the CNN model on multiple FPGAs. However, the resultant design may not be optimized for HW efficiency. By comparison, an RTL-based direct HW design may achieve higher efficiency and throughput, although it may require significant knowledge of both CNN model and FPGA architecture.

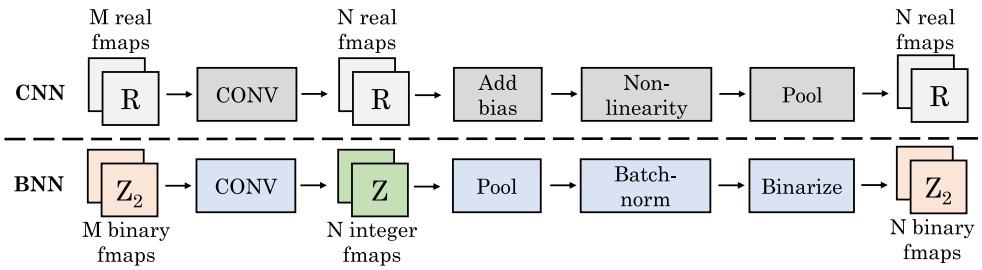


Fig. 5 A comparison of CNN and BNN [3, 6]

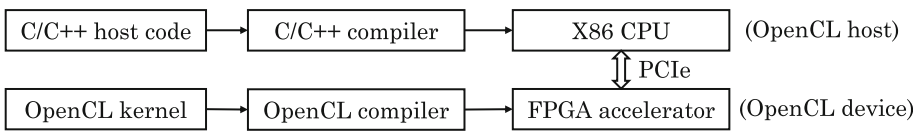


Fig. 6 The working approach of OpenCL-based FPGA accelerators

Limited HW resources of FPGA The compute and memory resources of FPGAs may be insufficient for CNNs used in real-life classification tasks. For example, the second layer of LeNet5 requires 2400 multipliers, which cannot be provided by most FPGAs [11]. Similarly, the BW of FPGA is much smaller than that of GPU ($\sim$ 20 versus 700GB/s) [7, 12–15]. While high-end FPGAs provide high BW, their input/output interface also incurs large area/power overheads.

The off-chip accesses can be reduced by using on-chip memory. In fact, for small NNs or on using memory optimization techniques (e.g., use of BNN), all the parameters may be stored in the on-chip memory itself [11, 16–18]. However, large on-chip memories are not energy efficient [19], and thus, the on-chip memory has limited effectiveness. These resource limitations pose severe design constraints. For example, limited HW resources may limit full unrolling of the loops, and hence, implementation of large CNNs on FPGA may require using temporal folding approach. This, however, may make it difficult for HLS tools to achieve efficient data layout and hardware partitioning.

Need of careful design and optimization Given the unique properties of both CNN models and FPGA architecture, a careful design is required for optimizing throughput. For example, in CONV layer, a large number of operations are preformed under nested loops. However, the dimension of every loop may not be much larger than the MAC units present on an FPGA. As such, if the MAC operation count is not perfectly divisible by the MAC unit count, some MAC units are unutilized which reduces resource usage efficiency. Due to this, different dimensions of MAC arrays can provide vastly distinct throughput for different layers and NN topologies [20]. In general, two designs with same resource utilization may have vastly different performances [21], and hence, resource utilization may not be a good indicator of overall performance.

The parameters of different CNN layers (e.g., kernel size, image size and number of fmaps) are vastly different. While reusing the HW module for different layers improves resource efficiency, it may lead to poor throughput due to a mismatch with computation patterns of different layers and the overhead of reconfiguration. Further, the BW utilization of FPGA depends crucially on the burst length of memory access [7], and hence, the memory access pattern of CNN needs to be carefully optimized. Moreover, the compute throughput should be matched with the memory BW; otherwise, the memory or compute may become the bottleneck in improving performance.

Relative merits of FPGAs vis-a-vis CPUs/GPUs/ASICs While the software frameworks for developing CNN models for GPUs/CPUs are already mature, those for FPGAs are still in early stages. Further, even with HLS tools, implementing CNN model on FPGAs may require multiple months for an expert HW designer [22]. Hence, in general, compared with FPGAs, GPUs provide higher performance with much lower design effort. Given the rapid progress in the field of NNs, re-architecting FPGA-based accelerator for every new NN algorithm may not be feasible. Hence, an FPGA-based design may not efficiently model the most recent NN algorithm. Further, due to the use of reconfigurable switches and logic blocks, an FPGA design incurs higher area/energy overhead than an ASIC design. These factors give a competitive disadvantage to FPGA vis-a-vis other computing platforms for HW acceleration of NNs.

CPU–FPGA heterogeneous computing State-of-the-art CNNs aimed at solving complex cognitive tasks use variety of techniques/operations. Implementing these on FPGAs may be inefficient or infeasible since they increase design time and complexity. To avoid these, such operations can be implemented on CPUs, and thus, only the computationally intensive operations may be implemented on FPGA. For FPGAs which do not share the memory with CPU, this leads to high data transfer overhead. Hence, CPU–FPGA shared memory systems need to be used [23]. Given the ease of programming CPUs, this also allows designing more versatile CNN accelerators. The limitation of these systems, however, is that they may not be widely available and may require higher expertise on the part of developer.

The techniques presented in remainder of this paper seek to address these challenges.

3 Hardware-friendly algorithmic optimizations

In this section, we discuss several algorithmic optimization strategies used in HW architectures for CNNs (Sects. 3.1–3.7).

3.1 Loop reordering, unrolling and pipelining

Loop reordering seeks to increase cache usage efficiency by avoiding redundant memory accesses between different loop iterations [21]. Loop unrolling and pipelining exploit parallelism between loop iterations by utilizing FPGA resources. For a loop with independent loop iterations, multiple iterations can be simultaneously executed which is termed as loop unrolling (or unfolding). Loop pipelining allows the next iteration to proceed even before an iteration has finished as soon as this iteration can provide data to the next iteration.

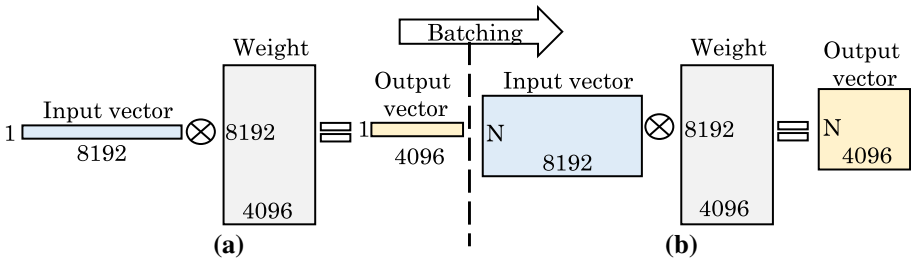


Fig. 7 Use of batching to convert **a** matrix–vector multiplication into **b** matrix–matrix multiplication

3.2 Tiling

For algorithms with large working set size, the data item fetched in cache may get evicted even before being used. To deal with this issue, tiling can be used which partitions the data fetched from memory into chunks (called tiles) that can fit in the cache [24]. This is to ensure that the data accessed in a compute region (e.g., loop) stay in the cache until they are reused.

Figure 3b shows the pseudo-code of an example CONV layer where four out of six loops are tiled. For example, tiling of input fmap means that instead of loading an entire fmap, only few rows and columns of that fmap are loaded, or instead of loading all fmaps, only few fmaps are loaded. The tiling of other loops (or data structures) can be similarly understood. Note that in this example, the kernel filter is not tiled, since the value of K (kernel filter size) is generally small. However, for a general case, the kernel can also be tiled.

The limitation of tiling is that it scrambles the data access pattern of memory. Assuming that fmaps are stored in row- or column-major format, on not using tiling, reading an fmap entails access to contiguous memory locations. This facilitates efficient memory reference using burst access. However, tiling over rows and columns interferes with this pattern which reduces the efficiency of memory access and memory BW. Thus, while improving cache utilization efficiency, tiling may harm memory access efficiency.

3.3 Batching

For sake of improving throughput, multiple input frames/images can be concurrently processed which is termed as batching. This is especially useful for improving reuse of weights in FC layers. Batching converts an MVM operation into an MM operation, as shown in Fig. 7a, b. Since batching may increase the latency, it is not suitable for latency-critical applications.

3.4 Fixed-point format

It is well known that on FPGA, FxP format incurs smaller area and latency overhead than FP format. In CNNs, use of nonlinear activation functions (e.g., `tanh` and `sigmoid`) already bounds the range of parameters, due to which use of FxP format does not harm the accuracy. Further, CNNs are generally used for error-tolerant applications. These factors provide significant scope and motivation for using FxP format in FPGA accelerators for CNNs, and hence, several works use FxP format in their designs.

3.5 Binarized CNNs

Compared with regular CNNs, BNNs are especially suitable for FPGA implementation due to the following reasons:

- Instead of 32b FP or 8b/16b FxP values, BNNs use 1b values which reduces their memory capacity and off-chip BW requirement. Also, binarized data have higher cacheability and can be stored on-chip. This is attractive given the low memory capacity of FPGAs. As an example, while the 8b FxP AlexNet requires 50MB memory for storing the parameters, the binarized AlexNet requires only 7.4MB memory [17] which allows storing the entire CNN in the on-chip RAM. Hence, on a typical FPGA, the FxP CNN becomes bound by the memory BW, whereas the BNN nearly reaches the peak of its computational performance [17].
- BNNs allow converting MAC operations of CONV layer into XNOR and population count (also called bit count) operations, as shown in Fig. 8a, b. Hence, instead of DSP blocks, XNOR gates can be used. Also, bit-count can be realized using an LUT, as shown in Fig. 8c. This leads to significant reduction in the area and energy consumption [17, 25]. Clearly, since FPGAs provide native support for bitwise operations, BNNs can be implemented efficiently. Further, with increasing CNN size, the accuracy of BNN approaches that of a 32b FP BNN [17].
- BNNs can allow FPGAs to become competitive with GPUs, as evident from the following example [16]. A Virtex-7 FPGA has 3600 DSP48 units, 433,200

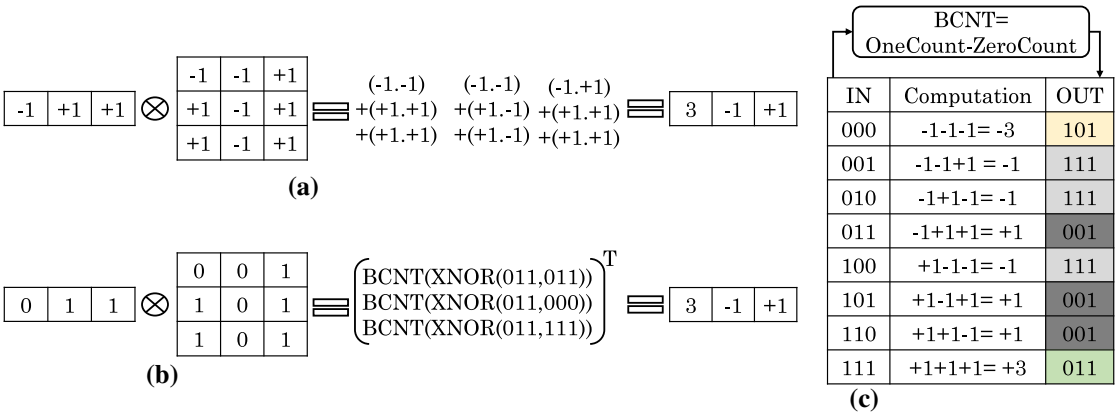


Fig. 8 **a** An example of binarized MM. **b** Implementing binarized MM using XNOR and BCNT (bit count) functions. − 1 is represented using 0. **c** Implementing binarized MM using an LUT (OUT is in 2's complement form) [26]

LUTs, whereas a Titan-X GPU has 3072 cores (one ALU per core). Their operational frequencies are 100–200 MHz and 1 GHz, respectively. We assume that an FxP and an FP MAC operation can be mapped to a DSP48 unit and a CUDA core, respectively [16]. Thus, despite comparable level of concurrency, the lower frequency of the FPGA makes its throughput much lower than that of the GPU for reduced precision CNNs.

By comparison, a BNN can be implemented more efficiently on an FPGA and a GPU. Assuming that a CUDA core can perform 32 bitwise operations in each cycle, a Titan-X GPU can provide $\sim$ 98,000-fold (= 3072 $\times$ 32) parallelism [16]. By comparison, on FPGA, bitwise operations can be executed even using LUTs. Assuming that a 6-input LUT can perform 2.5 XNOR operations, the Virtex-7 FPGA can provide $\sim$ 1,000,000-fold (= 2.5 $\times$ 433,000) parallelism [16]. Accounting for the frequency difference, FPGA and GPU can provide comparable level of throughput for the BNN.

The limitation of BNNs is that for comparable accuracy, they need 2–11 $\times$ larger number of parameters and operations [17]. Also, a BNN may still use 32b FP values for first and last layers. Due to this, both binary and 32b FP values need to be stored/processed which increases the design complexity.

3.6 HW-overhead reduction schemes

Given the high cost of implementing exact exponent function, some works use piecewise linear approximation of activation function [4, 27–30]. Similarly, to reduce the number of multiplications, some works use encoding [31] and Winograd algorithm [5, 32] or implement multiplications using logical elements (e.g., AND gates and half-tree adders) instead of DSP blocks [11]. In BNNs with {+ 1, − 1} weights/activations, padding with zero requires two bits of storage. To avoid this, Zhao et al. [3] and Fraser et al. [33] use + 1 padding and − 1 padding, respectively.

3.7 Spatial, frequency-domain and Winograd CONV

The spatial CONV (also called traditional CONV), which works by sliding the kernel over the image, performs high number of operations. To address these challenges, recent works have used Winograd algorithm and FDC.

In traditional (spatial) CONV, every element in the output fmap is separately computed. By comparison, in Winograd algorithm, a tile of output fmap is simultaneously generated based on the structural similarity between the elements in the same tile of the input fmap. Specifically, for a $N \times N$ input tile and $k \times k$ filter, Winograd algorithm produces an $J \times J$ output fmap, where $N = \mathrm{J} + k - 1$. For computing the neighboring J v J tile of output fmap, input tile needs to be shifted by J and Winograd algorithm can be applied again.

As for FDC, 2D CONV can be computed using 2D FFT, which, in turn, can be calculated as 1D FFT of every row followed by 1D FFT of every column. Use of 2D FFT lowers complexity of CONV layer; however, zero padding and stride lead to large number of operations, especially with small kernels. To further lower the complexity of discrete CONV of a long input with a small kernel, overlap-and-add (OaA) scheme can be used [5].

Winograd versus spatial CONV Compared with the traditional CONV, Winograd algorithm reduces the number of multiplications by reusing the intermediate outputs and is especially suitable for small kernel size and stride values. Some techniques decide about using spatial or Winograd CONV for each layer [34] since the Winograd

algorithm reduces computations but increases the BW pressure.

Winograd versus frequency-domain CONV Compared with Winograd algorithm, FFT-based approach (i.e., FDC) requires higher storage for storing transformed filter values [32]. For small filter sizes, FFT requires high number of operations due to the mismatch between the size of image and the filter. By comparison, for large filter sizes, Winograd algorithm incurs higher overhead due to addition and constant multiplications. Given that transformed filter values are in complex domain, the FFT approach has higher requirement of memory capacity and BW compared with the Winograd approach [32].

4 Classification

In this section, we first present the classification based on optimization strategies (Sect. 4.1) and then based on implementation and evaluation approaches (Sect. 4.2).

4.1 Classification based on optimization approaches

Table 1 shows the classification of the works based on the optimization approaches for CNN and FPGA architecture. Table 2 shows the works based on memory-related optimizations and use of well-known heuristics/algorithms.

4.2 Classification based on implementation and evaluation

Table 3 shows the classification of the works based on their implementation and evaluation approaches. Most research works accelerate inference phase which is expected since the training is performed relatively infrequently and the inference is performed on resource-constrained systems. Further, many works use high-level languages such as Caffe and OpenCL for reducing development time.

Apart from performance/throughput, many works also focus on saving energy which highlights the increasing importance of improving energy efficiency. Some works accelerate CNNs on multi-FPGA systems to be able to solve larger problems. Finally, many works compare their FPGA implementations with CPU/GPU/ASIC implementations to critically analyze the features of each platform.

5 CNN accelerator architectures

In this section, we discuss several NN accelerator architectures, e.g., fused-layer (Sect. 5.1), use of multiple CONV engines suited for different layers (Sect. 5.2) and systolic array architectures (Sect. 5.3). We then review accelerators which use multiple FPGAs (Sect. 5.4) or CPU–FPGA collaborative computing approach (Sect. 5.5).

Table 1 Optimization approaches

Strategy	References
Loop unrolling	[3–5, 7, 16, 20, 21, 24, 28–30, 35–41]
Loop tiling	[8, 19–22, 24, 35, 38, 42–48]
Loop re-ordering	[3, 20, 24, 36]
Pipelining	[4, 5, 7, 11, 16, 17, 21, 23, 25, 27–30, 32, 37, 40, 41, 49–53]
Batching	[16, 22, 26, 29, 41, 47, 49, 51, 54, 55]
Binarized CNN	[3, 17, 23, 25, 26, 33, 48, 56, 57]
Use of fixed-point format	[3, 4, 7, 8, 10, 11, 18, 22, 26, 28, 30, 36–38, 43, 44, 48, 58–66]
Per-layer quantization	[38, 44, 65]
Changing data layout in memory	[5, 32, 38, 44, 67]
Winograd algorithm	[32, 34, 41, 50, 52]
FDC	[5, 54]
Data compression	[68]
Prefetching	[8, 41, 45]
Approximating activation function	[4, 27–30]
Pruning	[38, 42, 46, 66]
Load balancing	[66]
Padding optimizations	[3, 33, 42, 54]
Eliminating FC layer	[57]

Table 2 Memory-related and other algorithmic optimizations

Strategy	References
Memory-related optimizations	
Double buffering	[5, 16, 21–24, 32, 41, 50, 59, 63]
Use of line buffer	[29, 34, 44, 48, 50]
Use of 3D memory	[64]
In-memory processing	[69]
Use of standard algorithms/models	
Roofline model	[7, 17, 21, 64, 68, 70]
Polyhedral optimization framework	[21, 43]
Greedy approach	[31, 44, 65]
Dynamic programming	[34, 40, 51]
Graph coloring	[22]
Graph partitioning	[53]
Singular value decomposition	[56]

Table 3 Classification based on implementation and evaluation approaches

Strategy	References
Accelerating inference versus training phase	
Inference phase	Nearly all
Training phase	[23, 47, 71]
CNN model description language/approach	
High-level languages/ synthesis approaches	Caffe [7, 34, 45, 52, 60, 67, 72], TensorFlow [22], OpenCL [4, 29, 64], C/C++ [3, 43], Java [73], others [3, 16, 17, 38]
RTL-HLS hybrid templates	[22]
RTL	[23, 35, 39]
Optimization target	
Latency or throughput	nearly all
Energy	[1, 3, 7, 10, 14, 16–19, 21–24, 26, 27, 32–34, 37, 38, 41, 44–46, 48, 49, 51, 55, 57, 58, 64–66, 68, 72, 74–77]
Evaluation	
Multi-FPGA system	[47, 51, 73]
Single FPGA	nearly all others
Systolic array architecture	[7, 23, 43]
CPU–FPGA collaborative computing	[5, 23, 45]
Comparison with	
CPU	[3–5, 7, 21, 22, 25–28, 37, 38, 44–47, 51, 52, 55, 61, 65, 66, 69, 70, 72, 76]
GPU	[1, 3, 7, 11, 16, 18, 22, 23, 25, 26, 28, 37, 38, 41, 44–47, 51–53, 55, 65, 66, 69, 72, 73, 75, 76]
ASIC	[11, 26]

5.1 Fused-layer architecture

Alwani et al. [78] note that processing the neighboring CNN layers one-after-another leads to high amount of off-chip transfers. They propose changing the computation pattern such that multiple CONV layers are computed together, which provides better opportunity of caching the intermediate fmaps and avoids their off-chip transfer. Their technique targets initial CONV layers of CNN, where majority of data transfers is in the form of fmap data. On fusing multiple CONV layers, only input fmaps of first layer are fetched from memory. During memory read, the interim values of all the fused layers that depend on the data being read are also computed. Only output fmaps of last layer are written to memory. In their technique, the first fused layer generates the outputs in the order that will be

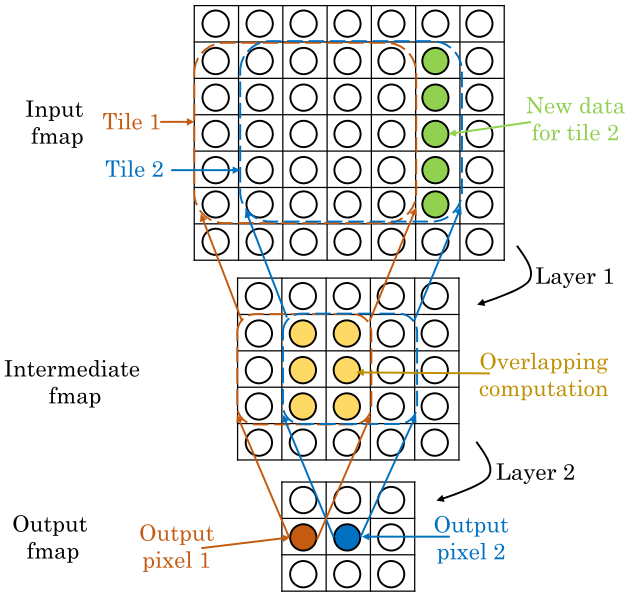


Fig. 9 An example of fusion of two CONV layers and formation of pyramid [78]

consumed by the next fused layer. Thus, the data can be directly passed to the next layer, without shuttling it off-chip.

Their technique is based on the observation of locality in the dataflow of CONV such that every point in a hidden layer depends on a specific portion of the original input fmaps. These inputs can be visualized as a pyramid across different layers of fmap, as shown in Fig. 9. Once an input tile is loaded on-chip, the entire pyramid of interim values is computed without loading/storing any extra fmap. On reaching the tip of pyramid (last fused layer), only the values in the last output fmaps are stored. After processing of one pyramid, a complete new input tile (pyramid base) need not be loaded. Instead, only one new column of the tile is loaded and one column of the tile is discarded.

Since two pyramids may overlap, some interim values may be used for computing both the output values. These values can be either (1) recomputed every time or (2) cached and reused. The first option leads to simpler implementation, whereas in second option, the computations performed by different pyramids become different. Their technique ascertains the pyramid dimensions and determines the storage/compute overheads of fusing along with its data transfer reduction benefits. Based on these and the amount of overlap between pyramids, the relative merit of option (1) or (2) can be ascertained. The pooling layer is always fused with CONV layer since it reduces BW without any overhead. With less than 400KB of on-chip memory, their design avoids off-chip transfer of more than 70MB of fmap data.

5.2 Layer-specific PE architectures

Shen et al. [40] note that FPGA-based accelerators use CONV layer engines (CLE) that process consecutive CNN layers one-at-a-time. However, since different CONV layers have different dimensions, using a fixed CLE dimension for all layers leads to poor utilization of resources and lower performance. Even though the CLE may be optimized to provide highest throughput, the CLE dimension may be sub-optimal for few and even most of the individual layers. For example, in Fig. 10a, the single-CLE HW operates on three layers (L1, L2 and L3) one-after-another. The dimension of CLE exceeds that of L1 which leaves some HW unutilized. However, CLE dimension is smaller than L3 dimension which requires using the CLE repeatedly to process different portions of L3. This again leaves some HW unutilized. They show that on AlexNet CNN, an optimal single-CLE may not utilize more than one-third of the resources. They propose a multi-CLE design, where the resources are partitioned across multiple smaller CLEs that closely match the layer dimension. These CLEs operate concurrently on different images, as shown in Fig. 10b. Here, CLE1 matches L1 and L3, whereas CLE2 matches L2, which improves HW usage efficiency.

They note that using one CLE for each layer is not efficient since it reduces the BRAM size available to each CLE. This also necessitates orchestrating many off-chip accesses which incurs BW and latency overheads. Also, the control overhead scales with the number of CLEs and

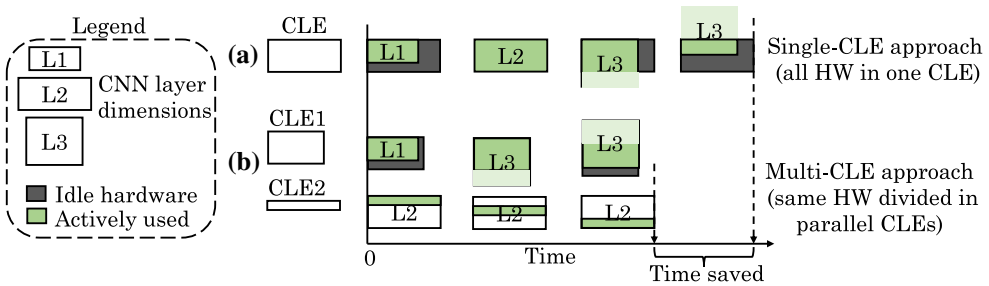


Fig. 10 A comparison of (a) single and (b) multi-CLE approach [40]. L1, L2 and L3 are CNN layers

leaves insufficient resources for CNN calculations. Instead, for an L-layer CNN, they propose using Q CLEs, where $Q < L$, under the constraint that the layer assigned to a CLE should have a matching dimension to optimize DSP utilization and the BRAMs allocated to CLEs should be such that the overall off-chip accesses are minimized. Their technique uses dynamic programming scheme to find the number of CLEs, resource-partitioning between them, and allocation and scheduling of layers from different inputs on the CLEs. Their design achieves higher throughput and resource utilization than single-CLE-based approaches.

5.3 Systolic array architecture

Wei et al. [43] propose a 2D systolic array (SA) architecture for implementing CNN. In every cycle, every PE shifts input and weight data to its adjacent vertical and horizontal PEs. Every PE has a SIMD vector accumulation module. Their design divides the global interconnect into local interconnects between adjacent PEs. Also, shifting of data between adjacent PEs eliminates muxes. Use of short, local interconnects allows their design to realize high frequency. To map a CNN model on SA, three steps are required: finding a feasible mapping, selecting a PE array configuration and ascertaining the data reuse approach. In the first step, a mapping is found which ensures that the proper data become available at desired locations in PE array in each cycle. In the second step, the size of each dimension of PE array is selected which influences clock frequency, DSP efficiency and DSP count. In the third step, the tile size is chosen to maximally exploit data reuse.

They develop an analytical model for resource utilization and performance and then explore the design space to find a design with maximum throughput. The complexity of search space is reduced by applying architectural constraints. The selected designs are then synthesized to find the clock frequency. Based on the actual frequency, the designs are further refined for finding the best systolic array configuration. Their technique translates a C program into CNN architecture using systolic array and achieves high throughput for FP and FxP data types.

5.4 Multi-FPGA architectures

Zhang et al. [51] note that due to its limited resources, a single-FPGA cannot provide high efficiency for mapping a CNN. They propose a deeply pipelined multi-FPGA design. Their optimization problem is to find the best mapping of every CNN layer to an FPGA to design a pipeline of K FPGAs, so as to maximize the overall throughput. They use dynamic programming to solve this problem in polynomial time. They use K channels for providing concurrent data streams to a SIMD computation engine, where K depends on the BRAM size and DSP count. Both FC and CONV layers are accelerated on a CONV engine. For CONV layers, they apply parallelism across input and output fmaps. In FC layers, weight matrix is shared by batching the input fmaps. Pooling and nonlinear layers are merged with CONV layer computations to reduce the latency. Thus, the number of channels in pooling and nonlinear layers is same as the output of CONV engines. They relate the arithmetic operation intensity with loop unrolling factors and find the unrolling factors for different layers to optimize performance.

They build a prototype with 7 FPGAs connected using fast and high-BW serial links in a ring network. Every FPGA runs a computation engine specific to one or more CNN layers. For avoiding overflow of the input buffer of every FPGA, a lightweight flow-control protocol is used. Their design achieves higher energy efficiency than multi-core CPU and GPU designs.

5.5 CPU-FPGA collaborative computing architectures

Moss et al. [23] accelerate a BNN on a CPU–FPGA platform. They use a systolic array design where every PE performs multiplication using XNOR and bit-count operations, as shown in Fig. 11. To leverage data use and reduce BW requirements, input vectors to every PE are interleaved. This necessitates use of a cache in each PE for recording the partial results, which are later accumulated. Two memory blocks, termed "feeders," are used on two boundaries of the systolic array which feed data to the grid. They implement their design on Intel Xeon + FPGA system. On this platform, FPGA can access Xeon's cache and the system memory can be addressed over a fast, high BW link. This allows running selected portions of algorithm on FPGA where FPGA is very efficient. Also, the sharing of address space between CPU and FPGA allows fine-grain task partitioning between them. For example, training a NN on a standalone FPGA is generally challenging; however,

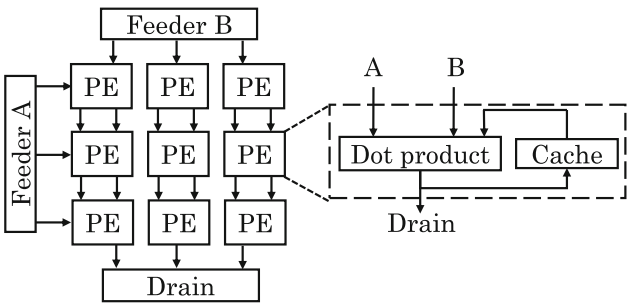


Fig. 11 Systolic array and PE design [23]

the Xeon + FPGA system allows performing NN training by using CPU for compute-intensive back-propagation steps. The FPGA implementation is achieved using RTL for maximizing performance. Their Xeon + FPGA design provides comparable throughput as the GPU design, while providing better throughput per watt.

Qiao et al. [45] present a technique for accelerating CNNs on FPGA. The FPGA has a dedicated module for accelerating MMs present in CONV and FC layers. They design a stream mapper unit for converting CONV operation to MM on-the-go, so that this conversion and MM can operate in parallel and hence the conversion overhead is eliminated. Since unrolled MMs in various CNNs have varied sizes, mapping them to fixed-configuration accelerator leads to sub-optimal performance. For efficiently performing MM of various sizes, they use tiling strategy. They also use stream prefetching to make the memory access stream sequential, which reduces ineffectual memory accesses and saves BW. Except CONV and FC layers, remaining tasks are performed on CPU (host). However, due to this, frequent data copies may be required between CPU and FPGA. To avoid this, they propose unified virtual memory approach, so that both CPU and FPGA can work in the same memory space. Their FPGA-based design achieves higher throughput/watt compared with both CPU and GPU and also other FPGA-based accelerators.

6 CNN simplification techniques

CNNs contain significant redundancy and are used for error-tolerant applications [79]. These facts allow significant simplification of CNN model which reduces its HW implementation overhead. This section discusses several techniques for reducing HW overhead of CNNs, e.g., pruning (Sect. 6.1), quantization (Sect. 6.2), use of BNNs (Sect. 6.3), encoding (Sect. 6.4), data compression (Sect. 6.5) and HW specialization (Sect. 6.6).

6.1 Pruning

Page et al. [46] present five CNN sparsification techniques for boosting CNN accelerator efficiency. First, they divide CONV layers with large filters into multiple layers with smaller filters, which effectively maintains the same receptive field region. For instance, a layer with 5×5 filters can be replaced with two successive layers, each with 3×3 layers. This reduces the number of parameters from 25 to 18, without much loss in accuracy. The filters can also be decomposed asymmetrically, e.g., a 3×3 filter can be decomposed into 3×1 and 1×3 filters, reducing the parameters from 9 to 6. Second, they represent CONV weights in FxP and FP formats with reduced bit width. Third technique, termed "feature compression partition," separates related feature channels which are processed in different branches of the computation graph. Thus, only relevant feature channels are supplied to fmaps of later layers of the network, as shown in Fig. 12b. For instance, at some stage of the network, if there are 256 feature channels, then they can be divided into four branches each beginning with a 1×1 CONV layer with 64 fmaps.

The fourth technique, termed "structured filter pruning," prunes the least important connections between input feature channels and fmaps, as shown in Fig. 12c. The fifth technique, termed "dynamic feature pruning," works on the observation that since inputs from different classes generally contain mutually exclusive abstract features, only few features are required for accurate classification. Hence, it dynamically prunes the feature channels of incoming data streams, as shown in Fig. 12d.

They further propose an FPGA accelerator for CNNs. For a given input channel, CONV over different output channels is performed in parallel. This approach reduces data transfer overhead compared with image patch tiling or parallelizing the CONV of input channels. Further, they use 16b FP format and also discuss strategies to implement their proposed sparsification techniques in HW. These techniques skip computation, memory access and data transfer for zero values. Their design achieves high

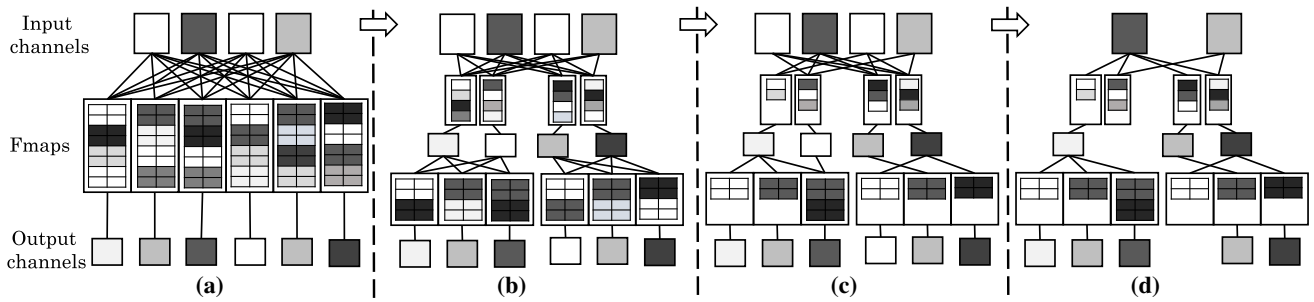


Fig. 12 **a** Original CONV layer. Illustration of successively applying three sparsification techniques to this CONV layer, **b** feature compression partition, **c** structured filter pruning and **d** dynamic feature pruning [46]

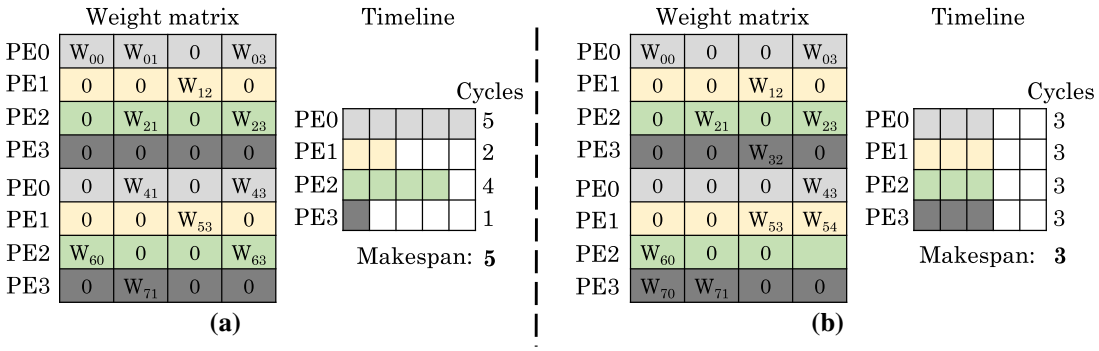


Fig. 13 A comparison of weight matrix and makespans in **a** naive pruning and **b** load-balance-aware pruning. Note that, naive pruning leads to high makespan whereas load-balance-aware pruning leads to short makespan

performance and energy efficiency with negligible impact on accuracy.

Kim et al. [42] design a CNN accelerator which is described in C and synthesized using an HLS tool. The spatial parallelism on FPGA is exploited using Pthreads. CONV, pooling and padding are implemented on FPGA, and remaining operations are performed on CPU. Since padding and pooling are tightly inter-related with CONV, they are also implemented in FPGA to reduce data transfer between FPGA and host CPU. Given the error tolerance of CNNs, they use 8-bit precision for performing computations. In their technique, output fmaps are calculated at tile granularity. An output fmap is computed in its entirety to avoid intermediate data transfer and compromising accuracy by rounding partial sums. At the end of a training phase, their technique stores nonzero weights and their intra-tile offsets. In the inference phase, the weight values and their intra-tile offsets are provided to the accelerator. Using this, in each cycle, a nonzero weight is processed and no cycles are wasted on zero weights. This zero-skipping approach improves the performance significantly.

In each cycle, one tile can be fetched from the on-chip RAM, and since four input fmap tiles are required for applying one weight tile, a minimum of four cycles are required for computing a weight tile. Hence, their zero-skipping scheme can provide a maximum of 75% (= 3/4) reduction in cycle count. The difference in the number of nonzero weights in concurrently computed output fmaps may lead to pipeline bubbles. To minimize its impact on efficiency, computation of different output fmaps is synchronized using a Pthreads barrier. Experiments confirm that their design achieves high throughput.

Han et al. [66] present a technique for improving inference speed by using load-balance-aware pruning, quantization and a custom hardware design. They prune weights which are smaller than a threshold and find that 90% weights can be pruned with negligible impact on accuracy. Pruning turns MM into sparse MM, where nonzero weights can be nonuniformly distributed to different PEs, leading to poor load balancing. To resolve this, they propose pruning in a way that sparsity ratio is nearly same in all the submatrices and this leads to better load balancing without harming accuracy. This is illustrated in Fig. 13.

After this, they quantize 32b FP weights to 12b integer. They use linear quantization approach for both weights and activations. For weight quantization, they analyze dynamic ranges of weights in all LSTM (long short-term memory) layers and then initialize the length of fractional part for avoiding data overflow. For activation quantization, optimal solution to the activation functions and intermediate outputs is searched. They use LUTs and linear interpolation and study the dynamic range of their inputs to determine the sampling approach. For example, for `tanh` activation function, the range of inputs is found to be [− 104, 107] and hence a sampling range of [− 128, 128] is used with 2048 sampling points. They note that on doing quantization, weights and indices are no longer byte aligned. Hence, they group 4b pointers with 12b weights to obtain 16b (2B) payload. Further, in pruned LSTM, a single element in the voice vector is consumed by several PEs which necessitates synchronization between the PEs. However, due to load imbalance, this leads to high completion time. Also, dynamic quantization increases the complexity of computations. They propose an FPGA design which addresses these challenges and can directly work on the compressed LSTM data. Their design achieves higher performance and energy efficiency than both CPU and GPU implementation.

6.2 Quantization

Qiu et al. [44] present a dynamic quantization approach and a data organization scheme for improving CNN efficiency. To boost FC layers, they use singular value decomposition which reduces the number of weights. They use dynamic quantization scheme where fractional length is different for different layers and fmaps but fixed in a single layer to avoid rounding-off errors. For fmap data

quantization, the intermediate data of FxP CNN and FP CNN are compared in layerwise manner using a greedy algorithm to minimize the loss in accuracy. They study different data quantization approaches for different networks. Compared with static quantization, their dynamic quantization scheme reduces storage while maintaining accuracy, e.g., using 8 and 4 bits for CONV and FC layers (respectively) reduces storage and BW consumption compared with using 16 bits for both layers. In their design, softmax function is implemented on CPU since it is used only in the last layer and its FPGA implementation incurs high overhead. Remaining operations are implemented on FPGA.

They exploit operator-level parallelism within CONV operations, intra-output parallelism across multiple CONV operations in every PE and inter-output parallelism by using multiple PEs. Also, to exploit reuse, they apply tiling on input image, input fmaps and output fmaps. Further, the tiles which are at same relative position in the image are stored contiguously. Thus, they can be loaded consecutively which saves BW. Their design provides high throughput and outperforms CPU- and embedded GPU-based implementations.

Park et al. [18] present an accelerator for speech and handwriting recognition which stores all the weights in FPGA memory only. Given the limited FPGA memory, they employ training-based weight quantization strategy and use 3-bit weights. They perform network training in three steps: conventional FP training, optimal uniform quantization to reduce the error in L2 norm and retraining with FxP weights. The parameters used in "training" and "retraining" are the same. The FxP weights of input and hidden layers are 3b, whereas that of output layer is 8b, since it shows higher sensitivity to quantization. To achieve a balance between resource usage and performance, they use a design where every layer has many PEs that operate in parallel, and each PE performs its serial processing in multiple cycles. Their architecture uses multiple tiles, and each tile, which has multiple PEs, implements one layer of the CNN. Their design achieves higher performance per watt compared with a GPU design.

Abdelouahab et al. [60] present a DSE approach for reducing DSP consumption in FPGA implementations of CNN. They use "CNN accuracy divided by DSP-count needed for its FPGA implementation" as the metric for measuring the improvement in accuracy brought by each DSP. Their technique controls two parameters, viz. the neuron count in each layer and precision of weights and biases. By changing these, their technique generates different HW designs with different topology and data representation of the network. By evaluating the entire design space, the design with highest value of the above metric is selected. Their technique reduces DSP utilization significantly while maintaining high classification accuracy.

6.3 Binarized CNNs

Zhao et al. [3] present an efficient FPGA implementation for a BNN. They note that in the BNN [6], although the weights are binarized, B-NORM parameters and biases are still real numbers. Since bias values are much smaller than 1 and inputs have a value of 1, they remove all biases and retrain the network. Further, they reduce the computation and storage requirement of B-NORM computation by noting that it is a linear transformation. They also quantized the FP inputs and B-NORM parameters of BNN to 20-bit and 16-bit FxP values, respectively.

They further note that in BNN, each activation can be $+1$ or -1, but the edge of input fmap is padded with 0 and hence, CONV can have 3 values: -1, 0 and $+1$. To resolve this issue, they use $+1$ padding to create a fully binarized CNN. Their design has three computation engines: FP-CONV engine for non-binary CONV layer 1, bin-CONV engine for binary CONV layers and bin-FC engine for binary FC layers. In FP-CONV unit, due to binary weights, multiplication can be replaced by sign inversion operation. The BIN-CONV unit accounts for the largest fraction of execution time. It performs CONV in two steps. In first step, input fmaps are read from the on-chip buffers and partial CONV sums are computed in CONV units and are accumulated in integer buffers. The input bits are suitably reordered to ensure that CONV units work for any fmap width and still utilize the HW efficiently. When all input fmaps in a layer are processed, the second step starts, where, pooling, B-NORM and binarization are performed. In Bin-FC unit, dot product between data and weights is performed by applying an XOR operation and adding the output bits using the population-count operation. For allowing multiple reads in each cycle, the data buffers are divided into multiple banks and fmaps are interleaved across different banks.

In BNN, the largest fmap for any layer is only 128K bits which can easily fit in on-chip memory of FPGA. They use double-buffering scheme, and thus, off-chip communication is required only for accessing input and weights and writing the output. Their design seeks to restrict the amount of integer-valued temporary data stored in buffers. For this, after transforming the binary inputs to integers in CONV/FC layers, other operations, viz. B-NORM, pooling and binarization, are also performed before writing the data to buffers. Their design provides higher performance/watt than a CPU, an embedded GPU and a server GPU.

Yonekawa et al. [25] note that BNNs require B-NORM for maintaining high accuracy which increases the area and memory accesses. They propose replacing B-NORM with

addition of an integer bias, and thus, B-NORM is approximated by an addition operation. This reduces HW complexity, area and memory accesses and allows exploiting higher amount of parallelism in CONV operations. The inputs, weights and outputs are stored in BRAMs which reduces power consumption compared with off-chip memory. Their design achieves higher performance/watt compared with embedded CPU and GPU and other FPGA implementations.

Umuroglu et al. [17] present a BNN accelerator based on heterogeneous streaming architecture. It uses different computing elements for each layer which transfer data through on-chip data streams. The computing elements are adapted for the needs of every layer. This boosts throughput although it incurs the overhead of requiring a different bitfile on a change in CNN topology. The BNN layers use B-NORM on outputs of CONV and FC layers and then use sign function for ascertaining the output activation. They realize this simply with unsigned comparison (i.e., thresholding) which requires much smaller number of LUTs, and no DSPs or flip-–flops. Then, max pooling and average pooling are realized using Boolean OR and majority function, respectively.

Their accelerator uses MVM-thresholding (MVMT) unit as the core computational unit, which can implement FC layer fully and is used as a component of CONV layers. MVMT has multiple PEs, each with multiple SIMD lanes. The weight matrix is stored in on-chip memory distributed across PEs. The input images stream through the MVMT unit where they are multiplied with the matrix. The design of a PE is shown in Fig. 14a. The dot product between input vector and a row of weight matrix is computed using XNOR gate. The number of set bits in the result is compared with a threshold to produce a 1-bit output.

CONVs are transformed to MM of filter matrix and image matrix for producing the output image. For this, filter matrix is generated by packing the filter weights and image matrix is generated by moving the sliding window over the input image, as shown in Fig. 14b. Since MVM forms bulk of the computation, folding it allows controlling the overall throughput. Folding of MVM is achieved by controlling the number of PEs and SIMD lane of each PE.

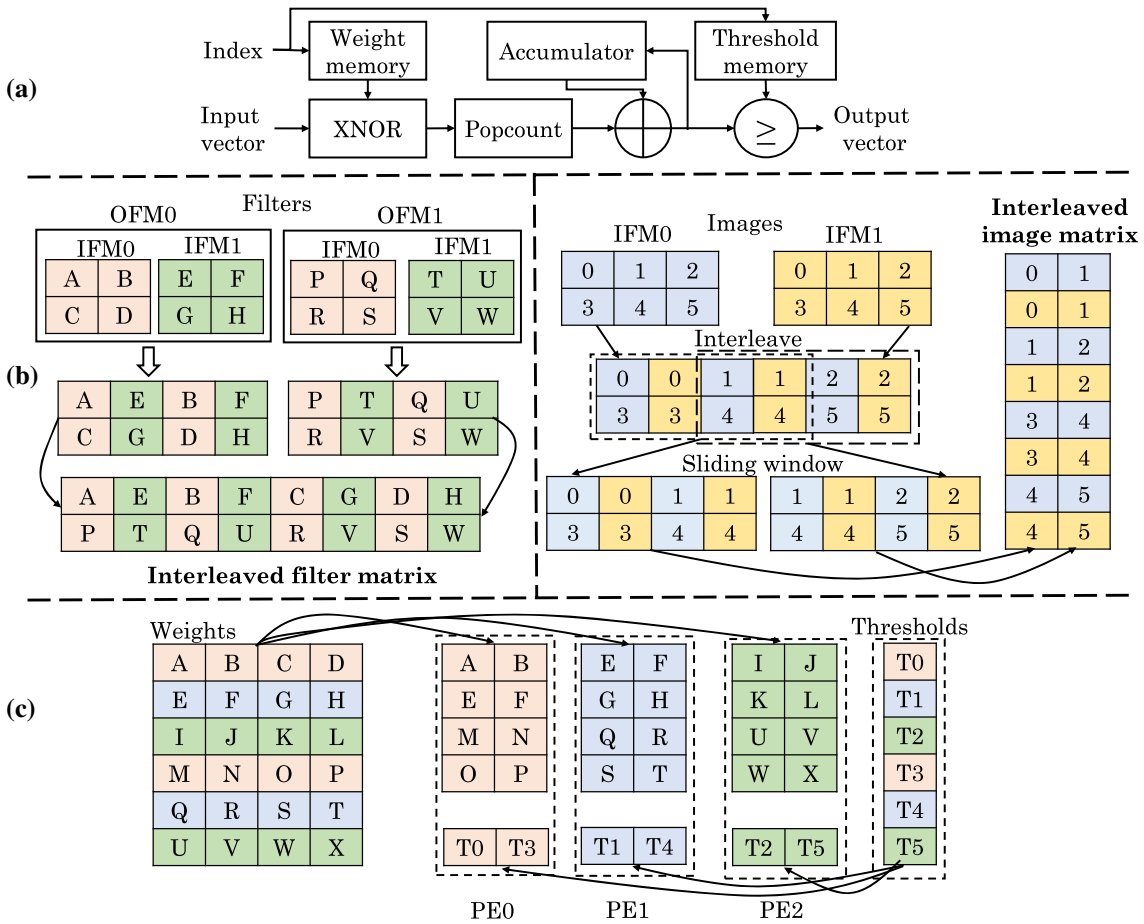


Fig. 14 **a** Datapath of PE of MVMT unit [17]. **b** Lowering CONV into MM (IFM/OFM = input/output fmap). **c** Folding MVMT on PE and SIMD lanes

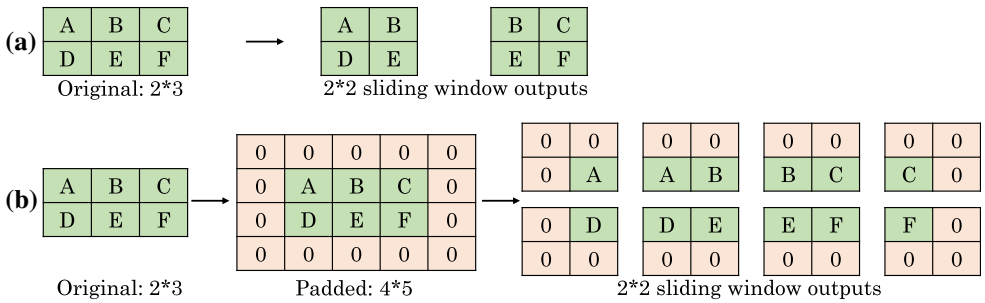


Fig. 15 CONV operation **a** without padding and **b** with padding [33]

This decides partitioning of the matrix across PEs. Every row and column of matrix tile are mapped to a distinct PE and SIMD lane, respectively. For example, as shown in Fig. 14c, with three PEs with two lanes, each MVM operation for a 6*4 matrix takes 4 (= (6/3)*(4/2)) cycles. They find the folding factor for layers such that every layer takes nearly same number of cycles and the target frame-per-second requirement is met. This avoids resource wastage due to a bottleneck layer. For small CNNs running on embedded FPGAs, their design provides high accuracy while incurring low latency.

Fraser et al. [33] note that in CONV layers of CNNs, zero padding is generally used for preventing loss of pixel information from image edges too quickly. As shown in Fig. 15, on using padding, pixels on the edge appear more frequently in the output of sliding window than that without padding. However, since BNNs have only $\pm$ 1 values but no zero value, use of ternary values ($\{-1,0,1\}$) will require two bits of storage which doubles the storage overhead. To avoid this, they use $-$ 1 padding and observe that it provides comparable accuracy as 0 padding. They use this strategy to scale the technique of Umuroglu et al. [17] to large CNNs which typically use padding. The resultant design provides high accuracy and throughput with low latency and power consumption.

Li et al. [16] accelerate BNNs on FPGAs using LUTs to perform computations for achieving high throughput. For first layer, the input data are scaled within the range [$-$ 31, 31] and are stored in 6b FxP format. Remaining layers work on binary fmaps. Further, since the parameters change only in training phase, but not inference phase, they merge binarization, B-NORM and value compensation in a modified sign function, viz. $y \geq c$, where y represents modified output fmap and c is a constant computed from normalization parameters. This function is termed as norm–binarize kernel. This requires storing only one threshold value (c) for every output value instead of multiple parameters. Also, both B-NORM and binarization can be realized using just one LUT-based comparator.

They parallelize the kernels over multiple PEs, each of which operates in SIMD fashion. PEs perform dot product by doing XNOR followed by bit counting for accumulation. PEs are realized using LUTs, as shown in Fig. 16. Accumulators are implemented using DSP48 slices. Weights and temporary results of accumulator outputs in a single fmap are saved in BRAMs. Fmaps are saved in distributed RAMs (registers). Since the maximum word length of a BRAM is 32 bits, the weights array is reshaped by 32 and then partitioned into multiple BRAMs to ensure high BW. By unrolling-dependent and independent loops, temporal and spatial parallelism (respectively) is exploited. To further exploit temporal parallelism and enhance resource utilization efficiency, loop pipelining is performed.

They compare their design with a GPU-based BNN design. On GPU, data-level interleaving is necessary for

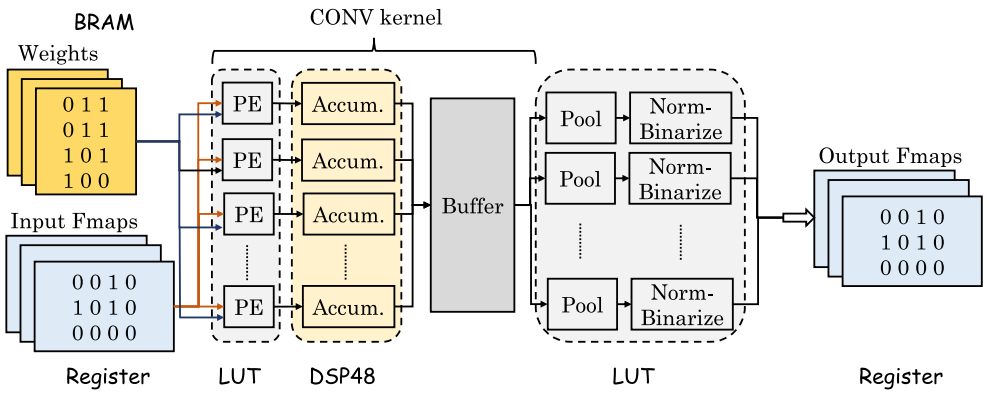


Fig. 16 The mapping of BNN on the FPGA resources in the design of Li et al. [16]

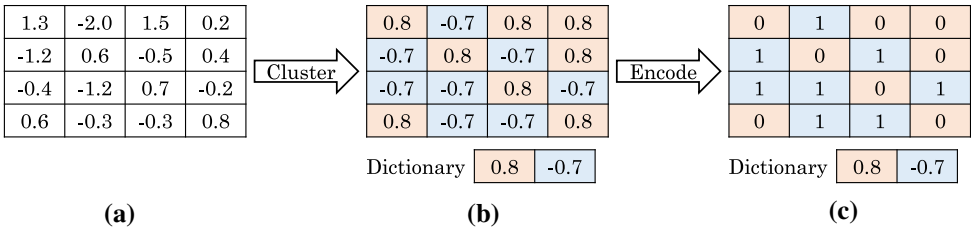


Fig. 17 Clustering and encoding schemes for reducing memory requirement of a CNN [31]. **a** Weight matrix with FP values, **b** matrix values replacedby centroids, **c** encoded matrix with 1-bitindex into dictionary

hiding the large latency of functional units, especially in the presence of loop-carried data dependency. Only in case of large batch size, thread-level parallelism can be leveraged for achieving high throughput. Unlike GPU, the throughput of FPGA is not sensitive to batch size. For processing individual requests in small batch sizes, their implementation provides higher throughput and energy efficiency than the GPU implementation. For large batch sizes, their throughput is comparable, but their FPGA design still provides higher energy efficiency than the GPU design.

Zhao et al. [47] present a reconfigurable streaming datapath architecture for training CNNs on FPGA. To implement training despite limited resources of FPGA, they divide training into multiple modules based on the layers. These modules are parameterized and can be specialized for CONV, pooling and FC layers. To lower the overhead of connecting modules, the modules support a unified streaming datapath. All the modules use the same input and output data layout which reduces BW wastage while accessing the memory. Their technique customizes the modules based on the layer configurations and partitions the training data into batches. Then, the following steps are iteratively performed: (1) a module is reconfigured into an FPGA device; (2) the data are loaded from CPU to DRAM for first layer or are already present in DRAM for subsequent layers; (3) the module is called to perform the computation; and (4) the result is read and weights are updated in the CPU.

Every module has one or more kernels, each of which executes basic operations, and its resource needs are guaranteed to fit within the resources of one FPGA. The kernels are reused with different weights and data, and different kernels can run concurrently. They discuss kernels for forward CONV, backward CONV, FC and pooling computations. They develop analytical models to capture the interconnection between BW, resource usage and performance, and also account for FPGA reconfiguration latency and data transfer latency between FPGAs. Based on these models, the configuration providing optimal performance is selected. Their design provides higher performance than CPU and higher energy efficiency than CPU and GPU.

6.4 Encoding

Samragh et al. [31] customize the CNN architecture for achieving efficient FPGA implementation. Figure 17 summarizes the working of their technique. Since FC layers are BW bound, they use encoding of CNN parameters which reduces data transfer from 32 bits to few (e.g., 2 or 3) bits. This alleviates off-chip BW bottleneck and may even allow storing the weights in BRAMs. Further, since MVM operations are compute bound, they propose factorized MVM of encoded values. This reduces the number of multiplications at the cost of higher number of additions, as shown in Fig. 18.

The encoding happens as follows. For a pre-trained network, their technique encodes its parameters depending on the memory budget and desired accuracy, using a greedy approach. Firstly, K-means clustering is performed on the parameters to minimize the distance between original parameters and their clustered values. Then, the loss in accuracy due to clustering is ascertained. If this exceeds a threshold, the error is reduced by increasing K in clustering algorithm and iteratively retraining the network. In retraining phase, the clustered values are fine-tuned for a fixed number of iterations. This retrained network has better robustness toward clustering since its parameters were initialized with clustered values. Their technique runs the encoding algorithm with $K \in \{2, \ldots, K_{\max}\}$ to select the least K value that reduces error rate below the threshold. The CNN is synthesized using parameterized HLS functions. The MVM performed in FC layers is parallelized

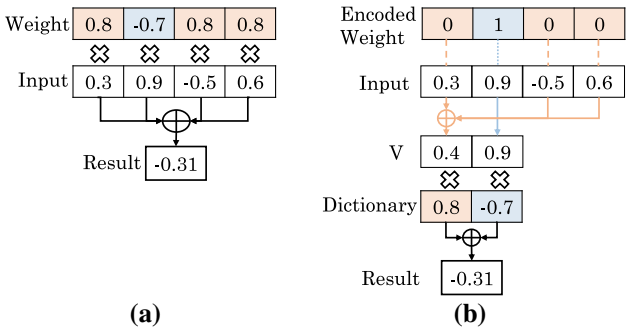


Fig. 18 A comparison of **a** traditional and **b** factorized dot product [31]

across rows. The rows are further parallelized by partitioning the input vectors and concurrently executing MAC operations on the partitions. The accumulated outputs of the partitions are added to obtain the overall output.

The "encoded" MVM kernel works similar to the normal MVM kernel, except that one of the MAC operands needs to be decoded before multiplying the values. The "factorized" MVM kernel also works similarly, except that it operates on encoded values and factorized coefficients. For these kernels, the parallelization factors for each layer are found with the consideration of balancing the latencies of each layer, minimizing resource utilization and maximizing throughput. To balance the latencies of the layers, higher resources are given to slow layers and the parallelization factors are again searched. Compared with baseline CNN implementation, their technique provides significant reduction in memory usage and increase in throughput with negligible loss of accuracy.

6.5 Data compression

Guan et al. [68] note that since CNN parameters are computed in training phase and remain unchanged in the testing phase, these parameters can be stored in compressed form and then decompressed at runtime. They present an FPGA design which uses data compression [80] to mitigate BW bottleneck of CNNs on FPGA. Since compression is performed offline, its latency overhead can be tolerated. Hence, they evaluate compression algorithms which provide high compression ratio, low overhead and latency of decompression. Specifically, they select LZ77 and Huffman encoding as examples of dictionary-based and entropy encoding-based algorithms, respectively. The number of decompression units determines the trade-off between their implementation overhead and BW reduction. As such, they perform DSE using roofline model to find the best allocation of on-chip resources between the CNN engine and decompressor unit for achieving highest possible performance with least BW. Their technique provides large improvement in performance.

6.6 HW specialization

Abdelouahab et al. [11] present a strategy for mapping a CNN to FPGA in unfolded manner. They model the CNN as a dataflow network which uses a data-driven execution scheme, such that the availability of inputs triggers the execution of the network. They map the entire graph on the FPGA to fully leverage CNN parallelism, i.e., the parallelism between neurons, CONV operations and multiplication operations of every CONV operation. This design obviates the need of off-chip accesses since the intermediate outputs need not be stored. The streams of fmaps are processed as they arrive and the execution is fully pipelined. To make their design feasible, they propose strategies to reduce its resource requirement. First, they use FxP representation for data and parameters with the minimal bit width to meet the accuracy targets. Second, the synthesis tool is forced to implement multiplications using logical elements in place of DSP blocks. The resultant implementation uses AND gates and trees of half adders.

They further note that one operand of multiplication operation, viz. the kernel, is constant during an inference phase. Hence, they specialize the multipliers to their constants. Although this necessitates re-synthesizing the design on each change in parameters values, it allows the synthesis tool to aggressively optimize low-level implementation. For example, multiplication with 0 can be avoided, that with 1 can be substituted by a straightforward connection and that with 2 can be replaced by shift operations. The experiments confirm that their FPGA accelerator provides high throughput.

7 Architectural optimization techniques

In this section, we review the research works in terms of their optimization techniques, such as loop tiling (Sect. 7.1), parallelization (Sect. 7.2), leveraging data reuse (Sect. 7.3), partitioning and folding (Sect. 7.4) and batching (Sect. 7.5). We then discuss techniques for optimizing spatial CONV (Sect. 7.6), frequency-domain CONV (Sect. 7.7) and Winograd CONV (Sect. 7.8). Finally, we discuss high-level tools for developing FPGA accelerators (Sect. 7.9).

7.1 Loop tiling

Zhang et al. [21] present an analytical approach for optimizing CNN performance on FPGA using roofline model. They use "polyhedral-based data dependence analysis" for finding valid variants of a CNN using different loop tile sizes and loop schedules. Since loop-carried dependence prohibits full pipelining of the loops, they use polyhedral-based optimization for permuting parallel loop levels to innermost levels to remove such dependences. After this, loop pipelining is applied. They apply loop tiling for loops whose loop bounds are large, since tiling the loops with small loop bounds (e.g., < 11) does not improve performance. Assuming that all the data required by PEs are stored on-chip, they find designs achieving highest performance (i.e., computation roof). However, due to memory BW limitations, these designs may not achieve the highest performance. To resolve this issue, they use "local memory promotion" strategy, whereby the loops which are

irrelevant to an array (i.e., the loop iterator does not appear in array access functions) are moved to outer loops.

For different loop schedules and tile sizes, they find "computation-to-communication ratio" which shows the operations performed for unit memory access. Then, among the designs whose BW requirement can be fulfilled by the given FPGA, the one with highest throughput is selected. In case of multiple such designs, the one with least BW requirement is selected since it will need the smallest number of I/O ports and LUTs, etc. They note that for different CONV layers of a CNN, the optimal unroll factor is different. However, such a HW design becomes highly complex due to the need of reconfiguring PEs and interconnections. To avoid this, they use the same unroll factor for all layers, which brings only small ($< 5\%$) loss in performance compared with a design with per-layer selection of unroll factor. Their technique provides higher performance and energy efficiency compared with CPU- and other FPGA-based designs.

Rahman et al. [20] present a DSE approach for evaluating different configurations of CNN accelerators on FPGAs. A processing array executes loop computation of the kernel expanded in some dimension of the iteration. These dimensions determine the shape of the compute array. They perform DSE across several dimensions, e.g., processing array shapes and sizes, loop tiling/unrolling/reordering. Under the constraints of DSP, RAM and off-chip memory BW resources, they seek to minimize execution cycle and, for equal cycles, minimize BW consumption. They note that the optimal shape of MAC array is crucially dependent on the CNN and to some degree on the FPGA. Their DSE approach enables finding high-performance architecture for different CNNs.

7.2 Parallelization

Motamedi et al. [70] note that in CNNs, although different layers cannot be executed in parallel, there is parallelism between different output fmaps ("inter-output parallelism"), between different CONVs ("inter-kernel parallelism") and between operations of a single CONV ("intra-kernel parallelism"). They propose a design which exploits all these parallelism. In their design, first layer has several multipliers that can be used together for CONV computations. The output of these multiplications is added using adder trees. The multipliers and adders together form a parallel CONV engine, and they allow leveraging intra-kernel parallelism. The output of CONV engines is added using adder stacks, and together, they allow leveraging inter-kernel parallelism. By using multiple such units, multiple output fmaps can be concurrently computed and thus inter-output parallelism is exploited. They use roofline model to find the optimal values of parameters (e.g., degree of parallelism) for optimizing throughput, while accounting for constraints, e.g., HW resources, memory BW and size of on-chip memory. By virtue of prudently exploiting all avenues of parallelism, their technique improves throughput and resource utilization efficiency significantly.

Moini et al. [10] present a technique for boosting CONV operations by maximally exploiting data reuse opportunities. In their technique, pixels at same spatial location of different output fmaps are concurrently computed. Hence, every pixel of input fmap and kernel data is accessed just once and saved in on-chip RAM until its reuse has exhausted. Out of the nested loops in CONV operation, the loop chosen for unrolling is one where the loop bound remains relatively large over different layers. For some loops, the loop bound reduces for later layers due to pooling and use of small kernel size. They implement their technique using FxP arithmetic. Their technique improves performance, energy and resource efficiency.

7.3 Exploiting data reuse

Zhang et al. [64] present an OpenCL-based architecture and design techniques for accelerating CNN on FPGA. They study the parameters of both the FPGA and the workload that impact performance, e.g., on-chip and off-chip BW, operations performed by the workload and data reuse behavior. Based on it, they note that if the memory BW requirement of a workload is larger than the BW provided by the FPGA, the workload is memory bound; otherwise, it is compute bound. They define "machine balance" as the match between memory BW and throughput of the machine (FPGA). Also, the "code balance" is defined as the memory BW required by a kernel of the workload. They study machine balance of different FPGAs and the code balance of a CNN. They observe that off-chip BW is a bottleneck for FC layers, but not for CONV layers; however, on-chip BW is a bottleneck for all the layers.

They note that a challenge in exploiting data-level parallelism by replicating PEs is that it leads to wasteful replication of memory, which consumes all the BRAMs. Based on the data reuse opportunities present in MM and the reconfigurable routing capabilities of FPGA, they use 2D connection between local memory and PEs, as shown in Fig. 19a. This allows several PEs to share data from same port of BRAMs. Thus, with no replication of memory, on-chip BW utilization and data reuse are improved. Further, instead of dispatching the work units to compute units in 1D manner (Fig. 19b), they are dispatched in 2D manner (Fig. 19c) using an adaptive scheduling scheme which balances memory BW with capacity. For FPGAs for which the data reuse requirement is still not met, they use a buffer which further exploits data reuse and coalesces memory

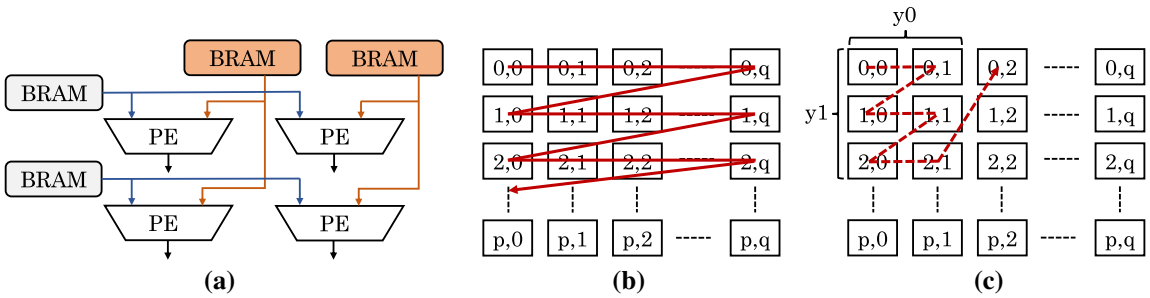


Fig. 19 **a** 2D interconnection between PE and local memory [64], **b** 1D dispatch, **c** 2D dispatch ($y0 = 2$, $y1 = 3$) [64]

accesses from multiple memory banks in shared memory to maximally use the off-chip BW.

They transform 2D CONV into MM by using "flatten and rearrange" strategy [4]. For this, they use a line buffer between on-chip and off-chip memory, as shown in Fig. 20. The line buffer converts continuous address stream from off-chip memory into the stream conducive for 2D CONV. This facilitates data reuse by increasing the locality and reduces overhead of random access from off-chip memory. Their design provides higher throughput compared with a CPU- and other FPGA-based implementations.

Wang et al. [29] present an FPGA-based CNN accelerator which performs pipelining of OpenCL kernels. Their design has four kernels connected via Altera's OpenCL extension pipes. The CONV kernel implements both MAC and inner product operation and hence is used in both CONV and FC layers. Pooling kernel does subsampling on the output data streams of the CONV kernel. MemRead and MemWrite kernels read/write feature data and weights from/to memory. These kernels operate in pipelined manner such that intermediate data need not be stored in global memory. The LRN function is not pipelined due to its different memory access pattern.

To improve pipeline utilization, a multi-mode CONV circuit with a pipelined multiply–add tree and delayed buffers is used. Also, input features and weights are vectorized, and different output fmaps are computed in parallel. To exploit data reuse, the input fmap data are replicated inside MemRead kernel to enable concurrent computation of output features. Also, the weights are fetched into a software-managed cache and are reused by different work groups which reduces memory accesses. For FC layer, multiple classifications are batched and operated in a single kernel. The pooling kernel uses a line-buffer-based design. It reads data of same fmaps one-line at-a-time and stores them in L line buffers (e.g., $L = 2$). Once the buffers are full, a window of fmap data is read and sent to the subsequent pooling stage. The exponent function of LRN kernel is implemented using the piecewise linear

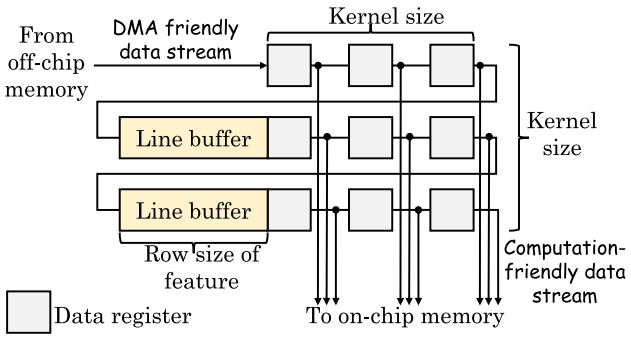


Fig. 20 The design of line buffer [64]

approximation approach [4]. Their technique achieves high performance with high resource utilization efficiency.

Motamedi et al. [8] present an FPGA-based CNN accelerator which exploits different sources of parallelism and low-precision FxP computations to boost throughput. Given limited on-chip memory, they use tiling to reduce off-chip accesses. Given the small size of kernel and input fmap row/column, tiling them is not required. Instead, they perform tiling so as to load selected input fmaps and kernels and store selected output fmaps. Tiling creates intermediate outputs; however, keeping them in on-chip memory reduces scope for reusing kernels and input fmaps, especially if the size of input fmaps exceeds that of intermediate outputs. Hence, they provide two options which maximize reuse of input fmap and output fmap, respectively. In first option, an input fmap is loaded only once and all CONVs that need this fmap must complete before evicting this fmap from on-chip memory. Some intermediate outputs may need to be written back to off-chip memory. The second option seeks to avoid this by loading input fmpas in innermost loops. Due to this, the kernels and input fmaps cannot be reused for obtaining output fmaps.

Recent FPGAs have several distributed but small BRAM units. Their design uses these to create multiple buffers to achieve high on-chip BW. Input/output fmaps and kernels are stored in different buffers. While CONV is being performed for one tile, next data tile is loaded in the neighboring buffer, which allows hiding the latency of off-

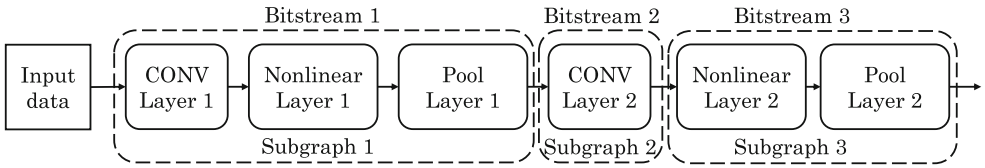


Fig. 21 An example of SDF partitioning [53]

chip access. They use multiple CONV engines for concurrently computing the pixel value using dot products. A parallel CONV engine is designed using multiple CONV engines, and it allows parallel (1) CONV of multiple input fmaps to compute one output fmaps and (2) computation of multiple output fmaps. FC layers are treated as CONV layers with a 1×1 kernel size. They perform DSE by changing the values of parameters, viz. parallelism at input/output fmap, the width of and number of ports. Based on this, the design which optimizes throughput under the constraints of BW, BRAM capacity and DSP blocks is found. Then, an RTL design is produced in Verilog for FPGA implementation. Their design provides high throughput.

7.4 Partitioning and folding

Venieris et al. [53] propose a domain-specific modelling approach and an automated design scheme for mapping CNNs on FPGA. They model CNN classification as a streaming problem which allows using "synchronous dataflow" (SDF) model of computation. This allows use of graph theory and linear algebra for reasoning about HW mapping of CNN. The programmer supplies the CNN model in high-level domain-specific format, along with the information about target FPGA. Then, the CNN is modeled in terms of DAG-based application model and platform-specific resource constraints are ascertained. Then, the CNN DAG is transformed into an SDF HW interim representation corresponding to fully parallel HW implementation. Every vertex of SDF graph represents a HW building block, and edges represent connections between them. Then, DSE is performed by applying graph transformations that do not cause functional change to the graph.

They use three types of transformations: (1) graph partitioning, (2) coarse-grain folding and (3) fine-grain folding. Graph partitioning splits SDF graph into subgraphs along CNN layers and maps each subgraph to a different bitstream, as shown in Fig. 21. On-chip memory is used for data reuse within the subgraph, and off-chip access is required only for input and output of the subgraph. This scheme requires FPGA reconfiguration when data flows to the next subgraph. To amortize this reconfiguration penalty, several input data streams can be computed in pipelined way. In coarse-grain folding, major operations of every layer (e.g., CONV, pooling, nonlinear operations) are fully unrolled. This provides highest throughput possible. In fine-grain folding, much smaller number of HW units (e.g., MAC) are used which are time-multiplexed between different operations. By changing the unroll factor and degree of multiplexing, a trade-off can be achieved between resource requirement and performance. Above-mentioned three transformations are used for producing a synthesizable Vivado HLS design for mapping on FPGA. Their approach improves performance density and achieves most of the performance of hand-tuned designs.

Venieris et al. [62] further note that latency-critical workloads cannot tolerate FPGA reconfiguration latency incurred due to graph partitioning scheme. To address this issue, they propose "weight reloading transformation." In this scheme, the graph is partitioned in multiple subgraphs and a single flexible architecture (and not different architectures) is produced which can execute all subgraphs by transitioning to different modes. This is shown as the "reference architecture" in Fig. 22. When data move to a new subgraph, their weights are read into on-chip RAM and appropriate datapath is formed by configuring the muxes. For example, in Fig. 22, the SDF of CNN is divided into three subgraphs, and the weight reloading is performed two times. The weight loading overhead depends on the weight count and system BW, but it is still smaller than the FPGA reconfiguration overhead.

Since the weights of even a single CONV layer can exceed the on-chip RAM capacity, they propose folding the input fmaps by a factor which is possibly different for each layer. Depending on the folding factor, a CONV layer is split into multiple subgraphs which execute a fraction of total CONVs. Finally, the interim results are accumulated to generate the output fmaps. Thus, the RAM requirement is reduced by the folding factor. Their technique allows optimizing for either throughput or latency. If the workload requirement can be fully met by the FPGA resources, the same design can provide both lowest latency and highest throughput. However, for large workloads, the weight reloading scheme along with latency optimization approach reduces the latency significantly while still maintaining a large fraction of the throughput of the throughput-optimization approach [53].

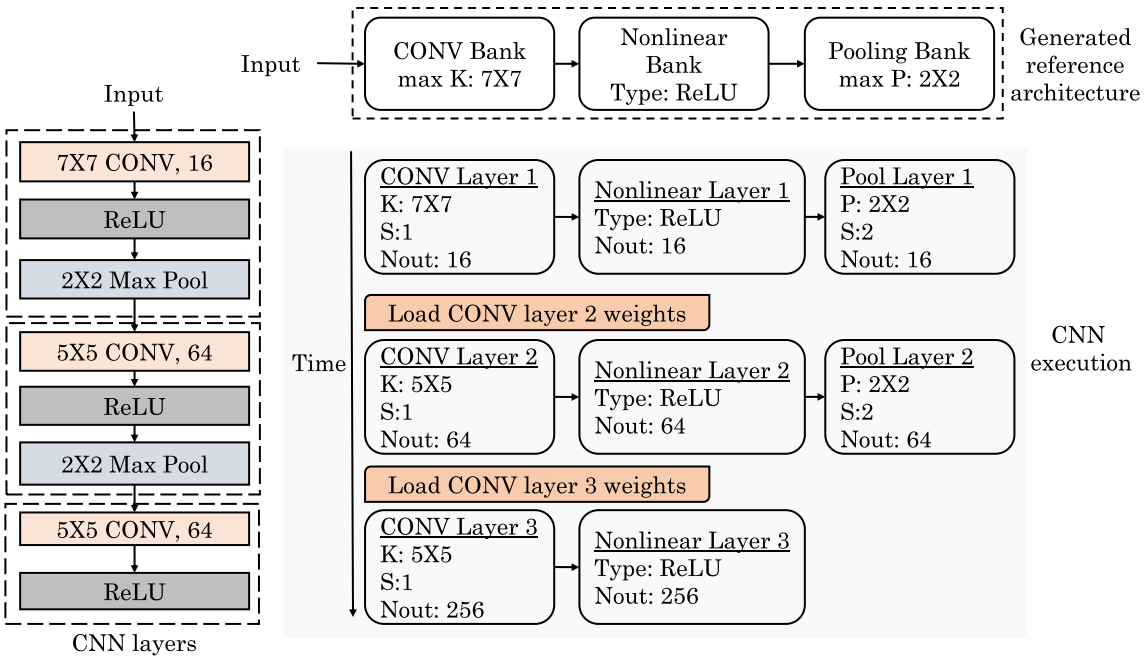


Fig. 22 An example of weight reloading [62] (*ReLU* rectified linear unit)

7.5 Batching

Shen et al. [63] note that effective use of batching requires carefully balancing input and weight transfer BW; however, conventional CNN accelerators do not support batching in CONV layers. They present an accelerator for reducing BW requirement of CNN by implementing adaptive data buffering. They first use a simple accelerator model with one adder and one multiplier. They note that for FC layers, with a fixed storage budget, increasing batch size requires reducing the data stored for each image. As weight reuse increases, BW required for weights reduces, but input data transfer also increases which now becomes a BW bottleneck. Overall, for a fixed amount of storage, decreasing the output data stored per image enables increasing the batch size, increasing input transfer per image by batch size and reducing the weight transfer per image by batch size. Further, batch size needs to be restricted to limit its impact on latency. They perform an exhaustive search to find the value of batch size for optimizing BW usage.

Reducing the BW requirement of FC layers shifts the BW bottleneck to the CONV layer, and hence, their technique allows batching of CONV layers also. As for CONV layer, it can be seen as a generalization of a FC layer, by substituting every input/output value with 2D fmaps and every weight with a 2D kernel. Thus, the constraints discussed above also apply for CONV layer. However, since MAC operations in FC layers generalize to 2D-CONV operations in CONV layers, the optimization problem becomes more complex. Specifically, along with batch size, the tiling factors of 2D inputs and outputs must also be considered, and they use exhaustive search to select the optimal values of these three parameters.

They further relax the assumption of one multiplier and one adder and model a realistic CNN accelerator which uses an array of vector dot product units and adders for leveraging parallelism. Overall, their design allows independently adjusting batch size, per-image data storage and tiling factor, for each layer. Their design reduces peak BW needs of both FC and CONV layers, without harming throughput or increasing FPGA resource utilization.

Nurvitadhi et al. [26] present an accelerator for BNNs which can perform all the operations required for BNNs. The network parameters such as weights, inputs/outputs and B-NORM constants are stored in the RAMs and are supplied to PEs for processing. The PE supports both full-precision FxP and binarized multiplication operations. In their design, 32 weights and neuron values are packed in 32-bit chunks. The multiplication of inputs with weights in a PE happens as follows. The inputs to the first BNN layer are in FxP format, and hence, a single FxP multiplication and addition takes place at a time. In subsequent layers, inputs and weights are binarized and hence 32 binarized MAC operations happen concurrently. This provides significant improvement in performance compared with the 32-bit representation. Except in first layer, the accumulated integer result is converted to FxP format for writing into temporary buffers. They implement their accelerator in

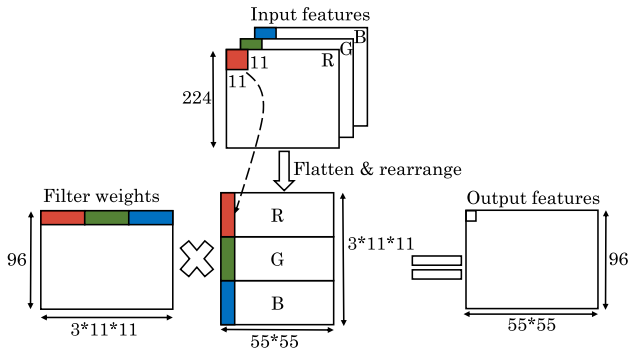


Fig. 23 An example of mapping 3D CONV to MM [4]

both FPGA and ASIC and compare them with CPU and GPU implementations.

They note that on not using batching, only one set of inputs is multiplied with the weights and hence fine-grained parallelism is not exploited. Hence, on both performance and performance/watt metric, CPU performs better than GPU due to extreme underutilization of GPU. On using batching, the utilization of both CPU and GPU increases and the performance of CPU and GPU becomes comparable. They note that binarization provides higher performance improvement than batching on both CPU and GPU. By combining both binarization and batching, even higher improvements can be obtained. Further, FPGA and ASIC provide one to two orders of magnitude higher performance than CPU and GPU by virtue of achieving high utilization, even though their peak performance is lower than that of GPU. Furthermore, FPGA and ASIC offer four and three (respectively) orders of magnitude improvement in energy efficiency compared with CPU.

7.6 Optimizing spatial CONV

Suda et al. [4] propose a DSE approach for maximizing CNN performance by leveraging reconfiguration capabilities of the FPGA. They use 8b and 10b precision for CONV weights and inner-product weights (respectively) which harms accuracy by $< 1\%$ in comparison with full-precision weights. For intermediate layer data, 16b precision is used. Using OpenCL, they develop CNN compute engines, e.g., an MM-based CONV module and other HW modules for B-NORM, pooling and FC layers. The CONV block is implemented by mapping 3D CONVs as MM operations by flattening and reorganizing the input features. For instance, the 224×224 CONV1 layer of AlexNet with three input features is reorganized to a matrix with dimension of $(3 \times 11 \times 11) \times (55 \times 55)$, as shown in Fig. 23. The features are stored in on-chip memory, and reorganization is performed at runtime which avoids off-chip accesses.

Pooling and FC layers are implemented as single work-item kernel, and they are accelerated by loop-unrolling and performing parallel MAC operations, respectively. Since the feature dimensions of different layers differ, they further use empirical and analytical models to perform DSE to optimize performance under the constraint of resource utilization, on-chip memory capacity and off-chip memory BW. DSE study finds optimal values of parameters such as loop-unroll factor and SIMD vectorization factor since they affect performance crucially. Evaluation of their technique with two CNNs on two different FPGAs shows that their technique achieves high throughput.

Zhang et al. [7] present a technique to boost performance of both FC and CONV layers. They note that transforming CONV MM to regular MM necessitates data duplication which increases the data size of input fmaps significantly (e.g., 25 $\times$). For avoiding data duplication, they propose converting regular MM in FC layer to CONV MM in CONV layer. They discuss two mapping schemes: input-major mapping (IMM) and weight-major mapping (WMM). In IMM, values from different input vectors in FC layer are mapped to the same input fmap in CONV layer. This improves data reuse of FC weight kernels during CONV of values from different images in the batched fmaps.

In WMM, weight vectors of FC layer map to input fmaps of CONV layer and input vectors of FC layer map to weight kernels of CONV layer. Weight vectors of FC layer are aligned in input fmaps such that weight values at the same position of all weight vectors are grouped into the same input fmap. Hence, every FC layer input can be reused multiple times. The reasoning behind WMM is that in FC layer, the weight kernel matrix is much larger than the input fmap matrix. Hence, WMM may allow higher data reuse than IMM given the limited HW resources. Both IMM and WMM improve BW utilization compared with the naive mapping, since they access contiguous memory locations. By using roofline model, they find that WMM allows higher data reuse and BW utilization during CONV, especially for small batch sizes. Hence, they use convolutional MM format for both FC and CONV layers and use WMM in FC layer. Mapping of FC layer to CONV layer using IMM or WMM allows using a uniform format for all cases. Their systolic-like FPGA accelerator is developed in HLS and allows changing parameters such as fmap size, precision (float, half or fixed) and number of PEs. The software interface allows changing parameters, e.g., number of layers, fmap sizes, shape and stride of weight kernels. Their design provides higher performance than CPU and higher energy efficiency than both CPU and GPU implementations.

Guan et al. [22] present a tool for generating FPGA-based accelerators from the CNN model described in

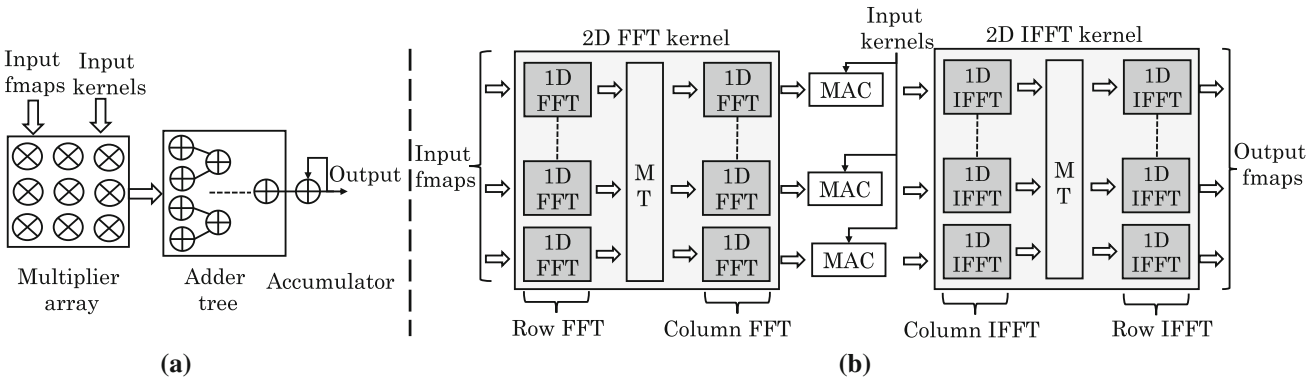


Fig. 24 A comparison of **a** direct and **b** OaA-based 2D CONV operations (*MT* matrix transpose) [5]

TensorFlow language. Their compiler converts the model described in TensorFlow to C++ program and FPGA programming bitstream, which are run by CPU and FPGA, respectively. The compiler works by extracting topology and operations of the target model. After optimizing and parameterizing the HW implementation, it produces the HW kernel schedule and kernel configuration. From the kernel schedule, C+ code is produced which is executed by the CPU (host). From the kernel configuration, FPGA (device) code is generated by instantiating RTL–HLS hybrid templates.

The model parameters and intermediate results are stored in off-chip memory due to limited BRAM. The compiler generates the execution graph. Given that FPGA has limited resources, only one kernel (e.g., CONV kernel) is allocated which performs inference for multiple layers. Also, to maximize the reuse of data buffers allocated in DRAM, the buffer reuse problem is modeled as graph coloring and then solved using a polynomial-time algorithm. Given the limitations and advantages of both pure-RTL and pure-HLS templates, hybrid RTL–HLS templates are used for HW generation which provide the best features of both. Here, RTL is used for developing a high-performance compute engine and OpenCL-based HLS for implementing control logic for RTL. From the kernel configuration produced by compiler, optimized kernel templates are produced which are used for generating HW codes for FPGA.

They split the operations of every layer into compute-intensive and layer-specific portions. Compute-intensive portions are processed using an MM kernel which is implemented in Verilog and is optimized using tiling. Since data access and computation patterns of different layers are different, they use different approaches for converting them into MM. For CONV and FC layers, they convert CONV and MVM into MM and then reuse the MM kernel. Similarly, they also discuss the approaches for pooling and activation layers.

To mitigate off-chip BW bottleneck, the memory data layout is changed in a manner to store successively accessed data at contiguous locations. This avoids redundant fetches and duplication of data. For FC layers, batching is used. They also use 16b FxP data values for saving resources. Their tool achieves higher performance than hand-coded FPGA accelerators. Also, their FPGA-based accelerator provides higher performance and energy efficiency than a CPU-based and higher energy efficiency than a GPU-based implementation.

7.7 Optimizing frequency-domain CONV

Zhang et al. [5] present a technique for accelerating CNNs on CPU–FPGA shared memory systems. Since 2D CONV is compute intensive, it is performed on FPGA and the remaining operations such as pooling, B-NORM, ReLU and FC layer are implemented on CPU. They use FFT and OaA for performing efficient CONV and implement frequency-domain algorithms on a OaA-based 2D CONV unit. Since the input fmap size far exceeds the typical kernel sizes ($m \times m$, where $m = 1, 3, 5, 7$), use of OaA is suitable in CNN. Their OaA-based 2D CONV unit has three modules: 2D FFT kernel, MAC (which can operate on complex numbers) and 2D IFFT kernel, as shown in Fig. 24. For every kernel, MAC operation is performed with all input tiles and the results are overlapped for obtaining one output fmap. By repeating this process for all kernels, complete output fmap is obtained. The CONV unit does not perform final overlapping since the overlapped stride width is decided by the kernel size which changes across layers. Since overlapping is a lightweight function, it is performed on CPU itself. They key benefit of using OaA is to lower the number of FP operations and boost data parallelism. The OaA CONV unit can work for any kernel size less than FFT size by using zero padding. For this, CPU performs data rearrangement for different CONV layers.

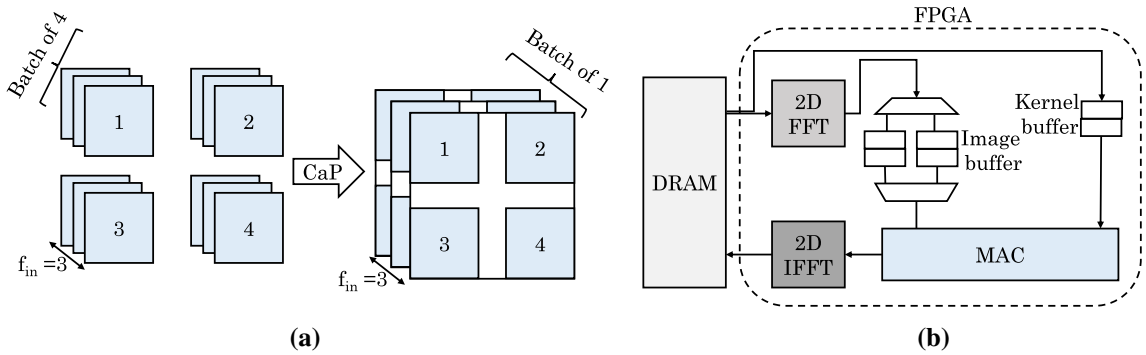


Fig. 25 **a** CaP operation [54], **b** overall system architecture proposed by Zeng et al. [54]

They note that if the data requirement of CONV unit exceeds the interconnect BW, the CONV unit is stalled. To avoid this, they propose reducing the data parallelism by folding the FFT kernel. If Q is the size of 1D FFT, they fold it by a factor of K ($1 \leq K \leq Q$) which reduces the data parallelism from Q^2 to Q^2/K. To achieve task parallelism, multiple CONV units are concurrently used. Since performing CONV with 1×1 kernel in frequency domain in inefficient, the CONV of 1×1 kernel is performed in time-domain only. Shared memory paradigm enables concurrent processing between CPU and FPGA. This allows reducing the data remapping latency across the layers. They also change the data layout in shared memory to improve its spatial locality and allow efficient data communication between CPU and FPGA. They use double buffering to maintain peak performance and lower memory access latency. Their technique improves performance while reducing the requirement of multipliers and memory.

Zeng et al. [54] optimize FPGA accelerator design by modifying FDC. They note that the baseline FDC reduces computation complexity under the assumption that FFT size equals image size and FFT is performed on the whole input image. However, while the image size varies greatly across the layers, the FFT size cannot be easily changed and supporting all FFT sizes leads to poor efficiency. Some works allow discrete FFT sizes at the cost of incurring reconfiguration overhead. The OaA scheme addresses this issue since tiling takes care of image scaling between layers. They use the OaA scheme for CNNs and demonstrate that small-sized FFTs reduce the computations for all the CONV layers. They further propose a hybrid technique where baseline FDC is used for small images and FDC with OaA is used for large images. This technique needs only small FFT sizes and reduces reconfiguration overhead while leveraging low-computation advantage of baseline FDC.

The limitation of first technique is that it requires image padding to match FFT sizes which leads to useless operations. Also, with limited FFT sizes, handling images of widely varying sizes is inefficient. Since changing FFT size incurs reconfiguration overhead, their second technique changes image size by reshaping the input data. They propose "concatenate and pad" (CaP) operation which is dual of OaA. In CaP, d^2 images of equal size are arranged into a $d \times d$ mesh such that there are x zero-value pixels between neighboring images. Here, x is the zero-padding size. Thus, CaP produces a large image by concatenating the original images with zero pixels, as shown in Fig. 25a. This technique applies both CaP and OaA to exercise control on the image size; specifically, the images in later layers are transformed by CaP. Thus, a single FFT size suffices for all layers. This avoids reconfiguration completely and also reduces the number of computations. The limitation of CaP is that it increases latency due to the use of batching. Their third technique combines baseline FDC, OaA and CaP–OaA. Figure 25b shows their overall system architecture. Their technique is effective in improving the latency or throughput of the overall CNN.

7.8 Optimizing Winograd CONV

Podili et al. [32] propose using Winograd minimal filtering algorithm [81] for reducing the number of multiplications in the CONV operation. Although this algorithm increases the number of additions, the overall cost is reduced due to higher relative overhead of the multiplier compared with the adder. This algorithm is effective in case of small filter size. They implement the 1D CONV engine in four-stage pipelined manner, as shown in Fig. 26a. The 2D CONV engine implements Winograd minimal 2D algorithm as a six-stage pipeline and uses 1D CONV engine as the building block. This is shown in Fig. 26b. Pipelining allows achieving high frequency which is determined only by the multiplier stage and not the adder stage.

By feeding the channels of a tile to the CONV engine in successive cycles and computing a running sum, their design avoids the need of temporary memory or adder trees. Thus, in subsequent cycles, next channels of the

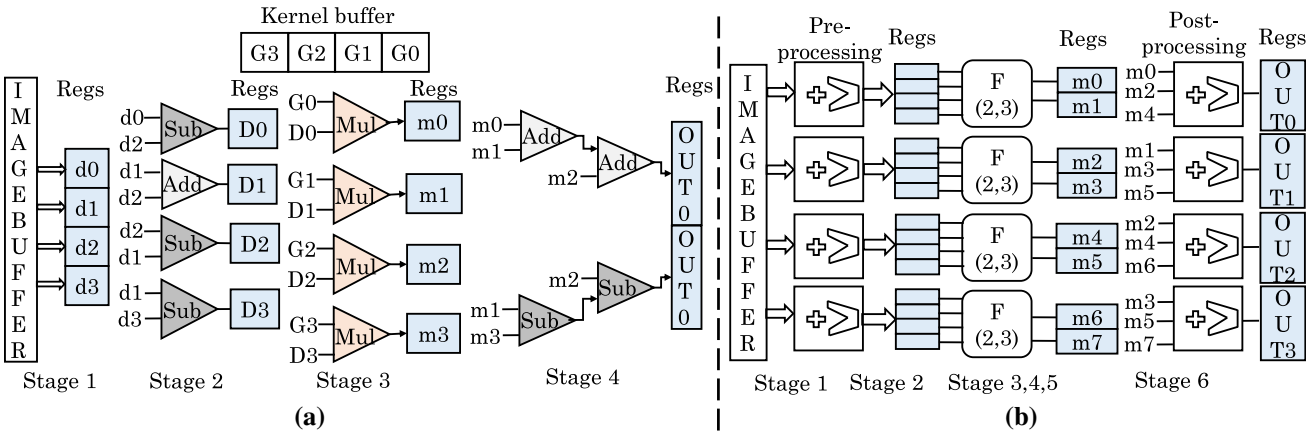


Fig. 26 **a** Datapath for $F(2,3)$, where $F(m,k)$ computes m outputs of an k-tap filter in Winograd algorithm [32] (regs = registers). **b** Datapath for $F(2*2,3*3)$.

present tile can be fed instead of next tiles of the present channel. Since CONV layer output matches the access pattern of the pooling layer, both CONV and pooling can be simultaneously performed in a pipelined manner. Thus, the latency of pooling can be easily hidden.

To improve resource efficiency, their technique aims to feed data to all CONV engines in every clock cycle. For this, extensive data reuse of kernel buffers is ensured and parallelism is exploited only across kernel buffers (and not the inputs). They also reorganize the data layout of on-chip buffers for reducing memory BW consumption. Their technique improves throughput while making efficient use of power and memory resources.

Aydonat et al. [41] present an architecture for boosting FPGA-based CNN implementations. To fully leverage FPGA HW, they use loop unrolling in CONV layer. The CONV loops chosen for unrolling are those that provide sufficient parallelism to fully utilize FPGA resources. Also, these loops should have high parallelism in a wide range of CNNs, and thus, using their design for different CNNs requires changing merely the unrolling factor and not the loop chosen for unrolling. The CONV layers execute on the HW in time-multiplexed fashion, i.e., in successive order. The input fmaps and weights are stored in on-chip RAM to exploit data reuse. Fmaps are stored in a double buffer, and weights are stored in the cache of PEs. Every PE receives the fmaps for computation and also passes the data to a neighboring PE, thus forming a daisy chain connection. Output fmaps are also stored in the double buffer. To avoid idling of PEs, at the time of reading the weights for one layer from the cache, the weights of next layer are pre-fetched into the cache.

Multiply–accumulate operations are simplified using Winograd transformation, which reduces the number of arithmetic operations and is feasible because of the use of small (3×3) filters in recent CNNs. They note that use of half-precision (16b) FP format reduces resource usage compared with use of single-precision (32b); however, 16b FP format is not natively supported on the FPGA. To lower this overhead, they use shared exponent scheme, which works on the observation that DSP blocks of their FPGA can be split into two 18×18 integer multipliers. They transfer both fmaps and filter data into 18b numbers using the maximum exponent present in the group. Now that all numbers have the same exponent, they can be taken as FxP numbers and can be directly fed to the 18×18 multipliers. After performing dot product, the number is converted to 10b and the exponent and sign bits are added to obtain 16b FP number. Since this scheme is performed only once before feeding the data to PEs, the same unit can be used for all the PEs.

The PEs used for CONV layers are also used for FC layers; however, since the FC layer does not show weight reuse, storing the weights in PE caches is not beneficial. To resolve this issue and exploit data reuse, they use batching in FC layers. After completion of CONV layers for an image, at the last stage, the result is dumped to the external memory for performing batching. Once a reasonable number of images become available, the batch is computed together in every FC layer. Overall, in CONV layer, features are shared which are multiplied with different weights to generate different output fmaps, whereas in FC layer, filter weights are shared which are multiplied with different image features to generate output features of different images. Hence, in FC layers, the weights are stored in the buffer and are streamed to PEs, and fmaps are stored in PE caches. Their design also supports pooling, activation and B-NORM layers, which can be optionally bypassed. All these units are specified as OpenCL kernels. Their design provides higher throughput than other FPGA implementations and higher throughput/watt than a GPU implementation.

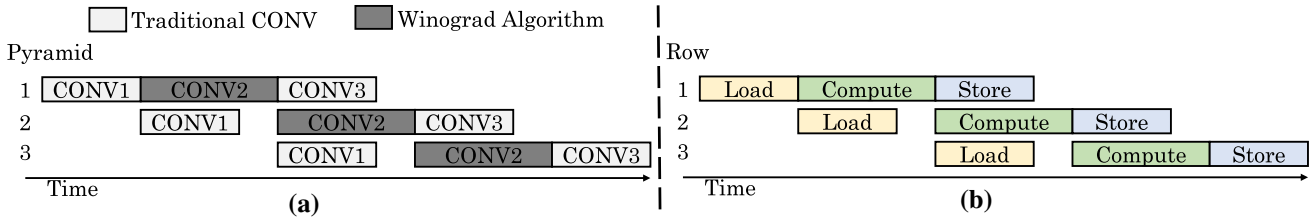


Fig. 27 **a** Inter-layer and **b** intra-layer pipeline [34]

Xiao et al. [34] present a technique which fuses multiple layers of CNN [78] to reduce computations and BW pressure. The overlap between the pyramids of neighboring elements provides scope for data reuse. To exploit this, they use circular line buffer for every layer, which achieves data reuse with lower complexity and less BRAM resources than the tile-based buffer used by Alwani et al. [78]. Execution of different layers, which are fused together, is pipelined, as shown in Fig. 27a. Within a layer, data load and store are overlapped with computation and this is shown in Fig. 27b. They note that compared with traditional CONV, Winograd algorithm reduces computations; however, it increases the BW pressure. To achieve the best of both, their technique decides about using traditional CONV or Winograd algorithm for each layer individually.

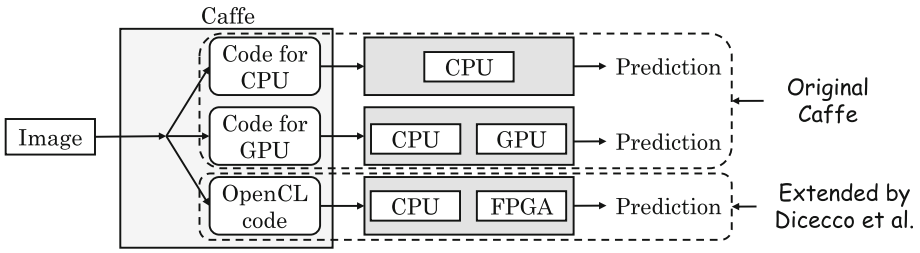
They model the optimization problem of deciding the division of CNN into fusion groups, the algorithm (traditional or Winograd) for every layer and its HW parallelism (i.e., number of processing units). The goal is to maximize throughput under the constraint of data transfer limit. This problem is solved using dynamic programming and branch-and-bound search. Based on these, HLS source code is generated using templates. They use templates for different layers, e.g., CONV (both traditional CONV and Winograd algorithm), LRN and pooling. To enable fusion of layers, they are wrapped into a function. For enabling pipelining of layers, dataflow is implemented across layers. The memory channels between layers are implemented as FIFO buffers. Finally, the source code is compiled into bitstream. Their technique provides high throughput and energy efficiency.

Dicecco et al. [52] adapt Caffe language for implementing a CNN on FPGA-based systems via OpenCL. Their design flow is illustrated in Fig. 28. Their technique allows specifying a program layer (for enabling reprogramming of FPGA), a pipelined layer and Winograd CONV engine. Pipelined layers pack multiple kernels in a single binary and perform transfer between kernels through local memory (FIFO). These layers reduce the requirement of reprogramming FPGA and synchronizing memory with the host between the layers. Further, pipelined layers increase throughput by allowing concurrent execution of layers. They use Winograd CONV algorithm with an output tile size of $m \times m$ and filter size of $r \times r$, where $m = 2$ and $r = 3$. They also modify this algorithm for reducing its DSP utilization. They apply Winograd CONV algorithm for any 3×3 CONV and obtain high throughput for several CNNs.

7.9 Accelerator development tools

Sharma et al. [72] present a technique for generating synthesizable accelerators for multiple FPGA platforms, based on the specification of a CNN in Caffe framework. Figure 29 shows the design flow of their technique. Their technique has four modules: translation, planning, design optimization and integration. First, the translator transforms the specified model to the macro dataflow ISA (instruction set architecture). Examples of instructions in the ISA are: `input`, `output`, `convolution` and `pool`. These instructions represent nodes in the dataflow graph of the CNN and are mapped to control signals for creating the execution sequence. This allows achieving optimizations specific to each layer and implementing a wide range of accelerators. Second, the planner module optimizes the HW templates for the target FPGA and balances data reuse and concurrency by partitioning computations and configuring the accelerator to match the FPGA constraints. The computations of every layer are partitioned into chunks such that these computations reuse the data by storing them in the local FIFO. Partitioning of computations allows overcoming the challenge due to limited FPGA memory. Between neighboring PEs, a unidirectional link is added to forward the shared input values for allowing data reuse. Their technique exposes concurrent execution of (1) PEs in a PU generating neighboring output elements and (2) PUs producing independent output fmaps. The planner module also generates a static execution schedule for the computation chunks.

Based on these, the designer module generates the accelerator core. This module uses multiple hand-tuned design templates and customizes them based on resource allocation and HW architecture determined by the planner. The templates have components for realizing various layers and a CNN that does not use a layer (e.g., B-NORM) can free corresponding resources for other layers. The designer



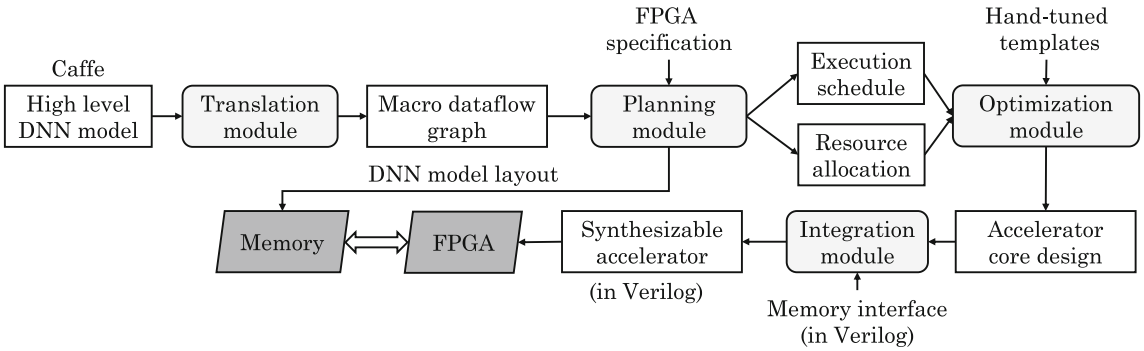


Fig. 28 Extending Caffe to generate FPGA code through OpenCL language [52]

Fig. 29 The overall flow of the technique of Sharma et al. [72]. For a CNN, it produces the synthesizable accelerator along with its execution schedule and memory layout

module produces state machines and microcodes based on the execution schedule. Finally, the integrator module glues memory interface code for a given FPGA with the accelerator code. Finally, the Verilog code is synthesized on the FPGA. Their FPGA-based accelerators provide higher efficiency and performance compared with GPU- and CPU-based implementations. Also, compared with HLS approach, their technique provides higher performance without requiring familiarity with HW design or programming frameworks.

Wang et al. [67] present a design automation tool which allows building accelerators for custom CNN configurations. Their tool includes an RTL-level accelerator generator and a compiler for generating dataflow under the user-specified constraints. By studying the basic operations performed by various CNNs, they determine the reconfigurable building blocks and record their HW description files and configuration scripts in a library. These blocks are not hardwired but allow configurability of different parameters, e.g., bit width and parallelism. By combining these blocks, the layers of most NN models can be realized, e.g., CONV layers and FC layers can be mapped to synergy neurons and accumulators; LRN layer and pooling layer can be mapped to LRN unit and pooling unit, respectively. The operations which cannot be mapped to logical gates are realized using an LUT, which is initialized by the compiler. If a requested key is not present in the LUT, its neighboring keys are checked and by interpolating those values, an approximate value for the requested key is generated.

The user provides the description of network topology and layers in Caffe format. If the NN cannot be fully implemented on FPGA due to its limited resources, it is implemented using two approaches: temporal folding, whereby multiple layers are mapped to common group of building blocks, and spatial folding, whereby a single layer is split into portions which share the building blocks at different times. The control-flow to synchronize different folds is generated by the compiler. The input fmap and weights are stored in memory in tiled manner to leverage data reuse through buffers. The address flow to access on-chip and off-chip memory is also generated by the compiler and is transformed into an FSM (finite state machine) which describes the memory access patterns. From the FSM, the RTL of address generator is produced. A coordinator connects the producer blocks to consumer blocks dynamically to form the datapath. Their work simplifies the development of accelerators for diverse range of NNs, e.g., multilayer perceptrons (MLPs), recurrent NNs (RNNs) and CNNs. Compared with other FPGA-based implementations, the FPGA-based accelerator generated by their technique shows higher energy efficiency.

Liu et al. [30] present a CNN accelerator along with a tool for automatically generating Verilog source code based on high-level descriptions of CNN layers. For CONV layer, they use a $K \times K$ multiplier array connected to an accumulate circuit. This unit can perform $K \times K$ MACs in

every cycle using pipelining. Right shifting of CONV window is achieved using a register. *K* BRAMs store input fmaps and can provide *K* pixels in each cycle. Using loop unrolling, they exploit intra-output and inter-output parallelism. For FC layer, they use a parallel multiplier array connected to an accumulation tree to perform dot product. They also implement pooling and normalization layers.

Their code generation tool extracts layer parameters from the high-level descriptions of CNN layers. Input/output fmaps are stored in 16b FxP, and intermediate values are stored in 18b FxP; however, since the range of values is different in different layers, the number of fractional/integer bits is selected separately for each layer. The bit width for weights is decided based on its impact on final accuracy. Based on these and the layer parameters, their tool generates Verilog source code which includes control logic and HW module instances, e.g., storage/processing modules. By analytically modeling the performance and resource usage of each layer, the optimal design is selected. Their technique provides high throughput and reduces FPGA development time.

8 Conclusion and future outlook

In this paper, we reviewed techniques for designing FPGA-based accelerators for CNNs. We classified the works based on several criterion to highlight their similarities and differences. We highlighted the prominent research directions and summarized the key ideas of different works. We close this paper with a brief mention of future research challenges.

FPGAs are optimized for integer and binary operations but show lower efficiency for FP operations. Transforming full-precision NNs to binary/quantized NNs may require significant training iterations, especially for complex cognitive tasks. Also, these low-precision NNs may not meet the accuracy requirement in mission-critical applications, such as medical and space. Given this, developing FPGA-based compute engines for full-precision networks is mandatory to extend the benefits of FPGA acceleration to a wide range of challenging scenarios.

Given the limited HW resources of a single FPGA, use of multiple FPGAs is essential for accelerating large-scale NN models. This requires partitioning the design across multiple FPGAs. While automated partitioning tools may not be widely available, manual partitioning approach is error prone and unscalable. Further, arbitrary splitting can increase interconnect complexity such that the number of connections may exceed the I/O pin count. For FPGAs to remain an attractive option for accelerating large-scale NNs, automated partitioning tools and resource-rich FPGAs are definitely required.

This paper focused on artificial neural network (ANN). In recent years, researchers have also explored spiking neural network (SNN) models which model the biological neuron more closely. SNNs and ANNs differ in significant ways, e.g., SNNs transmit information using spikes, whereas ANNs do this by firing rates. Due to this, their learning rules and network architecture also differ significantly. Going forward, modeling large-scale SNNs on FPGAs will present an exciting and rewarding challenge for computer architects [82].

Acknowledgements Support for this work was provided by Science and Engineering Research Board (SERB), India, Award Number ECR/2017/000622.

Compliance with ethical standards

Conflict of interest The author has no conflict of interest.

References

1. Ovtcharov K, Ruwase O, Kim J-Y, Fowers J, Strauss K, Chung ES (2015) Accelerating deep convolutional neural networks using specialized hardware. Microsoft Research Whitepaper vol 2, no 11
2. Mittal S, Vetter J (2015) A survey of methods for analyzing and improving GPU energy efficiency. ACM Comput Surv 47:19
3. Zhao R, Song W, Zhang W, Xing T, Lin J-H, Srivastava M B, Gupta R, Zhang Z (2017) Accelerating binarized convolutional neural networks with software-programmable FPGAs. In: FPGA, pp 15–24
4. Suda N, Chandra V, Dasika G, Mohanty A, Ma Y, Vrudhula S, Seo J-s, Cao Y (2016) Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In International symposium on field-programmable gate arrays, pp 16–25
5. Zhang C, Prasanna V (2017) Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system. In: International symposium on field-programmable gate arrays, pp 35–44
6. Courbariaux M, Hubara I, Soudry D, El-Yaniv R, Bengio Y (2016) Binarized neural networks: training deep neural networks with weights and activations constrained to + 1 or − 1. arXiv preprint arXiv:1602.02830
7. Zhang C, Fang Z, Zhou P, Pan P, Cong J (2016) Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks. In: International conference on computer-aided design (ICCAD), pp 1–8
8. Motamedi M, Gysel P, Ghiasi S (2017) PLACID: a platform for FPGA-based accelerator creation for DCNNs. ACM Trans Multimed Comput Commun Appl (TOMM) 13(4):62
9. Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, Erhan D, Vanhoucke V, Rabinovich A (2015) Going deeper with convolutions. In: IEEE conference on computer vision and pattern recognition, pp 1–9
10. Moini S, Alizadeh B, Emad M, Ebrahimpour R (2017) A resource-limited hardware accelerator for convolutional neural networks in embedded vision applications. IEEE Trans Circuits Syst II Express Briefs 64:1217–1221
11. Abdelouahab K, Pelcat M, Sérot J, Bourrasset C, Berry F (2017) Tactics to directly map CNN graphs on embedded FPGAs. IEEE Embed Syst Lett 9:113–116

12. Xilinx (2015) Ultrascale architecture FPGAs memory interface solutions v7.0. Technical Report
13. Mittal S (2014) A survey of techniques for managing and leveraging caches in GPUs. J Circuits Syst Comput (JCSC) 23(8):1430002
14. Chang AXM, Zaidy A, Gokhale V, Culurciello E (2017) Compiling deep learning models for custom hardware accelerators. arXiv preprint arXiv:1708.00117
15. Mittal S, Vetter J (2015) A survey of CPU–GPU heterogeneous computing techniques. ACM Comput Surv 47(4):69:1–69:35
16. Li Y, Liu Z, Xu K, Yu H, Ren F (2018) A GPU-outperforming FPGA accelerator architecture for binary convolutional neural networks. ACM J Emerg Technol Comput (JETC) 14:18
17. Umuroglu Y, Fraser NJ, Gambardella G, Blott M, Leong P, Jahre M, Vissers K (2017) FINN: a framework for fast, scalable binarized neural network inference. In: International symposium on field-programmable gate arrays, pp 65–74
18. Park J, Sung W (2016) FPGA based implementation of deep neural networks using on-chip memory only. In: International conference on acoustics, speech and signal processing (ICASSP), pp 1011–1015
19. Peemen M, Setio AA, Mesman B, Corporaal H (2013) Memory-centric accelerator design for convolutional neural networks. In: International conference on computer design (ICCD), pp 13–19
20. Rahman A, Oh S, Lee J, Choi K (2017) Design space exploration of FPGA accelerators for convolutional neural networks. In: 2017 Design, automation & test in Europe conference & exhibition (DATE). IEEE, pp 1147–1152
21. Zhang C, Li P, Sun G, Guan Y, Xiao B, Cong J (2015) Optimizing FPGA-based accelerator design for deep convolutional neural networks. In: International symposium on field-programmable gate arrays, pp 161–170
22. Guan Y, Liang H, Xu N, Wang W, Shi S, Chen X, Sun G, Zhang W, Cong J (2017) FP-DNN: an automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In: International symposium on field-programmable custom computing machines (FCCM), pp 152–159
23. Moss DJ, Nurvitadhi E, Sim J, Mishra A, Marr D, Subhaschandra S, Leong PH (2017) High performance binary neural networks on the Xeon + FPGA platform. In: International conference on field programmable logic and applications (FPL), pp 1–4
24. Ma Y, Cao Y, Vrudhula S, Seo J-s (2017) Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In: International symposium on field-programmable gate arrays, pp 45–54
25. Yonekawa H, Nakahara H (2017) On-chip memory based binarized convolutional deep neural network applying batch normalization free technique on an FPGA. In: IEEE international parallel and distributed processing symposium workshops (IPDPSW), pp 98–105
26. Nurvitadhi E, Sheffield D, Sim J, Mishra A, Venkatesh G, Marr D (2016) Accelerating binarized neural networks: comparison of FPGA, CPU, GPU, and ASIC. In: International conference on field-programmable technology (FPT), pp 77–84
27. Zhang Y, Wang C, Gong L, Lu Y, Sun F, Xu C, Li X, Zhou X (2017) A power-efficient accelerator based on FPGAs for LSTM network. In: International conference on cluster computing (CLUSTER), pp 629–630
28. Feng G, Hu Z, Chen S, Wu F (2016) Energy-efficient and high-throughput FPGA-based accelerator for convolutional neural networks. In: International conference on solid-state and integrated circuit technology (ICSICT), pp 624–626
29. Wang D, An J, Xu K (2016) PipeCNN: an OpenCL-based FPGA accelerator for large-scale convolution neuron networks. arXiv preprint arXiv:1611.02450
30. Liu Z, Dou Y, Jiang J, Xu J (2016) Automatic code generation of convolutional neural networks in FPGA implementation. In: International conference on field-programmable technology (FPT). IEEE, pp 61–68
31. Samragh M, Ghasemzadeh M, Koushanfar F (2017) Customizing neural networks for efficient FPGA implementation. In: International symposium on field-programmable custom computing machines (FCCM), pp 85–92
32. Podili A, Zhang C, Prasanna V (2017) Fast and efficient implementation of convolutional neural networks on FPGA. In: International conference on application-specific systems, architectures and processors (ASAP), pp 11–18
33. Fraser NJ, Umuroglu Y, Gambardella G, Blott M, Leong P, Jahre M, Vissers K (2017) Scaling binarized neural networks on reconfigurable logic. In: Workshop on parallel programming and run-time management techniques for many-core architectures and design tools and architectures for multicore embedded computing platforms (PARMA-DITAM), pp 25–30
34. Xiao Q, Liang Y, Lu L, Yan S, Tai Y-W (2017) Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs. In: Design automation conference, p 62
35. Ma Y, Cao Y, Vrudhula S, Seo J-s (2017) An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In: International conference on field programmable logic and applications (FPL), pp 1–8
36. Rahman A, Lee J, Choi K (2016) Efficient FPGA acceleration of convolutional neural networks using logical-3D compute array. In: Design, automation & test in Europe(DATE), pp 1393–1398
37. Liu Z, Dou Y, Jiang J, Xu J, Li S, Zhou Y, Xu Y (2017) Throughput-optimized FPGA accelerator for deep convolutional neural networks. ACM Trans Reconfig Technol Syst (TRETS) 10(3):17
38. Zhang X, Liu X, Ramachandran A, Zhuge C, Tang S, Ouyang P, Cheng Z, Rupnow K, Chen D (2017) High-performance video content recognition with long-term recurrent convolutional network for FPGA. In: International conference on field programmable logic and applications (FPL), pp 1–4
39. Ma Y, Suda N, Cao Y, Seo J-s, Vrudhula S (2016) Scalable and modularized RTL compilation of convolutional neural networks onto FPGA. In: International conference on field programmable logic and applications (FPL), pp 1–8
40. Shen Y, Ferdman M, Milder P (2017) Maximizing cnn accelerator efficiency through resource partitioning. In: International symposium on computer architecture, ser. ISCA '17, pp 535–547
41. Aydonat U, O'Connell S, Capalija D, Ling AC, Chiu GR (2017) An OpenCL deep learning accelerator on Arria 10. In: FPGA
42. Kim JH, Grady B, Lian R, Brothers J, Anderson JH (2017) FPGA-based CNN inference accelerator synthesized from multi-threaded C software. In: IEEE SOCC
43. Wei X, Yu CH, Zhang P, Chen Y, Wang Y, Hu H, Liang Y, Cong J (2017) Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In: Design automation conference (DAC), pp 1–6
44. Qiu J, Wang J, Yao S, Guo K, Li B, Zhou E, Yu J, Tang T, Xu N, Song S et al (2016) Going deeper with embedded FPGA platform for convolutional neural network. In: International symposium on field-programmable gate arrays, pp 26–35
45. Qiao Y, Shen J, Xiao T, Yang Q, Wen M, Zhang C (2017) FPGA-accelerated deep convolutional neural networks for high throughput and energy efficiency. Concurr Comput Pract Exp 29(20):e3850
46. Page A, Jafari A, Shea C, Mohsenin T (2017) SPARCNet: a hardware accelerator for efficient deployment of sparse convolutional networks. ACM J Emerg Technol Comput Syst (JETC) 13(3):31
47. Zhao W, Fu H, Luk W, Yu T, Wang S, Feng B, Ma Y, Yang G (2016) F-CNN: an FPGA-based framework for training convolutional neural networks. In: International conference on application-specific systems, architectures and processors (ASAP), pp 107–114

48. Liang S, Yin S, Liu L, Luk W, Wei S (2018) FP-BNN: binarized neural network on FPGA. Neurocomputing 275:1072–1086
49. Natale G, Bacis M, Santambrogio MD (2017) On how to design dataflow FPGA-based accelerators for convolutional neural networks. In: IEEE computer society annual symposium on VLSI (ISVLSI), pp 639–644
50. Lu L, Liang Y, Xiao Q, Yan S (2017) Evaluating fast algorithms for convolutional neural networks on FPGAs. In: International symposium on field-programmable custom computing machines (FCCM), pp 101–108
51. Zhang C, Wu D, Sun J, Sun G, Luo G, Cong J (2016) Energy-efficient CNN implementation on a deeply pipelined FPGA cluster. In: International symposium on low power electronics and design, pp 326–331
52. DiCecco R, Lacey G, Vasiljevic J, Chow P, Taylor G, Areibi S (2016) Caffeinated FPGAs: FPGA framework for convolutional neural networks. In: International conference on field-programmable technology (FPT), pp 265–268
53. Venieris SI, Bouganis C-S (2016) fpgaConvNet: a framework for mapping convolutional neural networks on FPGAs. In: International symposium on field-programmable custom computing machines (FCCM), pp 40–47
54. Zeng H, Chen R, Prasanna VK (2017) Optimizing frequency domain implementation of CNNs on FPGAs. Technical report
55. Li H, Fan X, Jiao L, Cao W, Zhou X, Wang L (2016) A high performance FPGA-based accelerator for large-scale convolutional neural networks. In: 2016 26th International conference on field programmable logic and applications (FPL). IEEE, pp 1–9
56. Lin J-H, Xing T, Zhao R, Zhang Z, Srivastava M, Tu Z, Gupta RK (2017) Binarized convolutional neural networks with separable filters for efficient hardware acceleration. In: Computer vision and pattern recognition workshop (CVPRW)
57. Nakahara H, Fujii T, Sato S (2017) A fully connected layer elimination for a binarized convolutional neural network on an FPGA. In: International conference on field programmable logic and applications (FPL), pp 1–4
58. Jiao L, Luo C, Cao W, Zhou X, Wang L (2017) Accelerating low bit-width convolutional neural networks with embedded FPGA. In: International conference on field programmable logic and applications (FPL), pp 1–4
59. Meloni P, Deriu G, Conti F, Loi I, Raffo L, Benini L (2016) Curbing the roofline: a scalable and flexible architecture for CNNs on FPGA. In: ACM international conference on computing frontiers, pp 376–383
60. Abdelouahab K, Bourrasset C, Pelcat M, Berry F, Quinton J-C, Serot J (2016) A holistic approach for optimizing DSP block utilization of a CNN implementation on FPGA. In: International conference on distributed smart camera, pp 69–75
61. Gankidi PR, Thangavelautham J (2017) FPGA architecture for deep learning and its application to planetary robotics. In: IEEE aerospace conference, pp 1–9
62. Venieris SI, Bouganis C-S (2017) Latency-driven design for FPGA-based convolutional neural networks. In: International conference on field programmable logic and applications (FPL), pp 1–8
63. Shen Y, Ferdman M, Milder P (2017) Escher: a CNN accelerator with flexible buffering to minimize off-chip transfer. In: International symposium on field-programmable custom computing machines (FCCM)
64. Zhang J, Li J (2017) Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In: FPGA, pp 25–34
65. Guo K, Sui L, Qiu J, Yao S, Han S, Wang Y, Yang H (2016) Angel-eye: a complete design flow for mapping CNN onto customized hardware. In: IEEE computer society annual symposium on VLSI (ISVLSI), pp 24–29
66. Han S, Kang J, Mao H, Hu Y, Li X, Li Y, Xie D, Luo H, Yao S, Wang Y et al (2017) ESE: efficient speech recognition engine with sparse LSTM on FPGA. In: FPGA, pp 75–84
67. Wang Y, Xu J, Han Y, Li H, Li X (2016) DeepBurning: automatic generation of FPGA-based learning accelerators for the neural network family. In: Design automation conference (DAC). IEEE, pp 1–6
68. Guan Y, Xu N, Zhang C, Yuan Z, Cong J (2017) Using data compression for optimizing FPGA-based convolutional neural network accelerators. In: International workshop on advanced parallel processing technologies, pp 14–26
69. Cadambi S, Majumdar A, Becchi M, Chakradhar S, Graf HP (2010) A programmable parallel accelerator for learning and classification. In: International conference on parallel architectures and compilation techniques, pp 273–284
70. Motamedi M, Gysel P, Akella V, Ghiasi S (2016) Design space exploration of FPGA-based deep convolutional neural networks. In: Asia and South Pacific design automation conference (ASP-DAC), pp 575–580
71. Han X, Zhou D, Wang S, Kimura S (2016) CNN-MERP: an FPGA-based memory-efficient reconfigurable processor for forward and backward propagation of convolutional neural networks. In: International conference on computer design (ICCD), pp 320–327
72. Sharma H, Park J, Mahajan D, Amaro E, Kim JK, Shao C, Mishra A, Esmaeilzadeh H (2016) From high-level deep neural models to FPGAs. In: International symposium on microarchitecture (MICRO). IEEE, pp 1–12
73. Baskin C, Liss N, Mendelson A, Zheltonozhskii E (2017) Streaming architecture for large-scale quantized neural networks on an FPGA-based dataflow platform. arXiv preprint arXiv:1708.00052
74. Gokhale V, Zaidy A, Chang AXM, Culurciello E (2017) Snowflake: an efficient hardware accelerator for convolutional neural networks. In: IEEE international symposium on circuits and systems (ISCAS), pp 1–4
75. Lee M, Hwang K, Park J, Choi S, Shin S, Sung W (2016) "FPGA-based low-power speech recognition with recurrent neural networks. In: International workshop on signal processing systems (SiPS), pp 230–235
76. Mahajan D, Park J, Amaro E, Sharma H, Yazdanbakhsh A, Kim JK, Esmaeilzadeh H (2016) Tabla: a unified template-based framework for accelerating statistical machine learning. In: International symposium on high performance computer architecture (HPCA). IEEE, pp 14–26
77. Prost-Boucle A, Bourge A, Pétrot F, Alemdar H, Caldwell N, Leroy V (2017) Scalable high-performance architecture for convolutional ternary neural networks on FPGA. In: International conference on field programmable logic and applications (FPL), pp 1–7
78. Alwani M, Chen H, Ferdman M, Milder P (2016) Fused-layer CNN accelerators. In: International symposium on microarchitecture (MICRO), pp 1–12
79. Mittal S (2016) A survey of techniques for approximate computing. ACM Comput Surv 48(4):62:1–62:33
80. Mittal S, Vetter J (2016) A survey of architectural approaches for data compression in cache and main memory systems. IEEE Trans Parallel Distrib Syst (TPDS) 27:1524–1536
81. Winograd S (1980) Arithmetic complexity of computations, vol 33. SIAM, Philadelphia
82. Maguire LP, McGinnity TM, Glackin B, Ghani A, Belatreche A, Harkin J (2007) Challenges for large-scale implementations of spiking neural networks on FPGAs. Neurocomputing 71(1):13–29