HAL
open science

Accelerating CNN inference on FPGAs: A Survey

Kamel Abdelouahab, Maxime Pelcat, François Berry, Jocelyn Sérot

To cite this version:

Kamel Abdelouahab, Maxime Pelcat, François Berry, Jocelyn Sérot. Accelerating CNN inference on FPGAs: A Survey. 2018. hal-01695375v2

HAL Id: hal-01695375

https://hal.science/hal-01695375v2

Preprint submitted on 13 Mar 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Accelerating CNN inference on FPGAs: A Survey

Kamel Abdelouahab[1], Maxime Pelcat[1,2], Jocelyn Sérot[1], and François Berry[1]

[1]Institut Pascal,Clermont Ferrand, France
[2]IETR, INSA Rennes, France

January 2018

Abstract

Convolutional Neural Networks (CNNs) are currently adopted to solve an ever greater number of problems, ranging from speech recognition to image classification and segmentation. The large amount of processing required by CNNs calls for dedicated and tailored hardware support methods. Moreover, CNN workloads have a streaming nature, well suited to reconfigurable hardware architectures such as FPGAs.

The amount and diversity of research on the subject of CNN FPGA acceleration within the last 3 years demonstrates the tremendous industrial and academic interest. This paper presents a state-of-the-art of CNN inference accelerators over FPGAs. The computational workloads, their parallelism and the involved memory accesses are analyzed. At the level of neurons, optimizations of the convolutional and fully connected layers are explained and the performances of the different methods compared. At the network level, approximate computing and datapath optimization methods are covered and state-of-the-art approaches compared. The methods and tools investigated in this survey represent the recent trends in FPGA CNN inference accelerators and will fuel the future advances on efficient hardware deep learning.

1 Introduction

The exponential growth of big data during the last decade motivates for innovative methods to extract high semantic information from raw sensor data such as videos, images and speech sequences. Among the proposed methods, Convolutional Neural Networks (CNNs) [1] have become the *de-facto* standard by delivering near-human accuracy in many applications related to machine vision (e.g classification [2], detection [3], segmentation [4]) and speech recognition [5].

This performance comes at the price of a large computational cost as CNNs require up to 38 GOP/s to classify a single frame [6]. As a result, dedicated hardware is required to accelerate their execution. Graphics Processing Units (GPUs), are the most widely used platform to implement CNNs as they offer the best performance in terms of pure computational throughput, reaching up 11 TFLOP/s [7]. Nevertheless, in terms of power consumption, Field-Programmable Gate Array (FPGA) solutions are known to be more energy efficient (vs GPUs). As a result, numerous FPGA-Based CNN accelerators have been proposed, targeting both High Performance Computing (HPC) data-centers [8] and embedded applications [9].

While GPU implementations have demonstrated state-of-the-art computational performance, CNN acceleration is shortly moving towards FPGAs for two reasons. First, recent improvements in FPGA technology put FPGA performance within striking distance to GPUs with a reported performance of 9.2 TFLOP/s for the latter [10]. Second, recent trends in CNN development increase the sparsity of CNNs and use extreme compact data types. These trends favorize FPGA devices which are designed to handle irregular parallelism and custom data types. As a result, next generation CNN accelerators are expected to deliver up to x5.4 better computational throughput than GPUs. [7].

As an inflection point in the development of CNN accelerators might be near, we conduct a survey on FPGA-Based CNN accelerators. While a similar survey can be found in [11], we focus in this paper on the recent techniques that were not covered in the previous works. Moreover, a recent review of efficient processing techniques for deep learning is proposed in [12], but focuses on Application Specific Integrated Circuits (ASIC) accelerators for CNNs while our work is mainly related to FPGA-based implementations.

The rest of the paper is organized as follows, section 2 recalls the main features of CNNs, focusing on computations and workload issues. Section 3 studies the computational transforms exploited to accelerate CNNs on FPGAs. Section 4 reviews the contributions that attempt to optimize the data-path of FPGA-Based CNN accelerators. Section 5 shows how approximate computing is a key in the acceleration of CNNs on FPGAs and overviews the main contributions implementing these techniques. Finally, section 6 concludes the paper.

2 Background on CNNs

This section overviews the main features of CNNs and focuses on the computations and parallelism patterns involved during their inference.

2.1 General Overview:

CNNs are feed-forward, deep, sparsely connected neural networks that implement weight sharing. A typical CNN structure consists of a pipeline of layers. Each layer inputs a set of data, known as a Feature Map (FM), and produces a new set of FMs with *higher-level semantics*.

2.2 Inference *vs* Training:

As typical Machine Learning (ML) algorithms, CNNs are deployed in two phases. First, the *training* stage works on a known set of annotated data samples to create a model with a *modeling* power (i.e. which semantics extrapolates to natural data outside the training set). This phase implements the *back-propagation* algorithm [13]

which iteratively updates CNN parameters such as convolution weights to improve the predictive power of the model. CNN Models can also be *fine-tuned*. When *fine-tuning* a model, weights of a previously-trained network are used to initialize the parameters of a new training. These weights are then adjusted for a new constrain, such as a different dataset or a reduced precision.

The second phase, known as *inference*, uses the learned model to classify new data samples (i.e inputs that were not previously seen by the model). In a typical setup, CNNs are trained/fine-tuned only once, on large GPU/FPGA clusters. By contrast, the inference is implemented each time a new data sample has to be classified. As a consequence, the literature mostly focuses on accelerating the inference phase. As a result, this paper overviews the main methods employed to accelerate the inference[1]. Moreover, since most of the CNN accelerators benchmark their performance on models trained for image classification, we focus on this paper on this application. Nonetheless, the methods studied in this survey can be employed to accelerate CNNs for other applications such object detection, image segmentation and speech recognition.

2.3 Inference of CNNs

CNN inference refers to the *feed-forward* propagation of B input images across L layers. This section details the computations involved in the major types of these layers. A common practice is to manipulate layer parameters and FMs using tensors. The tensors and variables used in this work are listed in table 1.

Table 1: Tensors Involved in the inference of a given layer ℓ with their dimensions

$\boldsymbol{X}$	Input FMs	$B \times C \times H \times W$	B	Batch size (Number of input frames)
$\boldsymbol{Y}$	Output FMs	$B \times N \times V \times U$	$W/H/C$	Width / Height / Depth of Input FMs
$\boldsymbol{\Theta}$	Learned Filters	$N \times C \times J \times K$	$U/V/N$	Width / Height / Depth of Output FMs
β	Learned biases	N	K/J	Horizontal / Vertical Kernel size

2.3.1 Convolution layers:

A convolution layer (*conv*) carries out the feature extraction process by applying –as illustrated in figure 1– a set of 3D-convolution filters $\boldsymbol{\Theta}^{\text{conv}}$ to a set of B input volumes $\boldsymbol{X}^{\text{conv}}$. Each input volume has a depth C and can be a color image (in the case of the first *conv* layer), or an output generated by previous layers in the network. Applying a 3D-filter to 3D-input results in a 2D *Feature Map (FM)* and, each *conv* layer outputs a set of N two-dimensional features maps. In some CNN models, a learned offset β^{conv} –called a *bias*– is added to the 3D-conv results, but this practice is discarded in recent models [6]. The computations involved in feed-forward propagation of *conv* layers are detailed in equation 1.

$$\forall \{b, n, u, v\} \in [1, B] \times [1, N] \times [1, V] \times [1, U]$$
$$\boldsymbol{Y}^{\text{conv}}[b, n, v, u] = \beta^{\text{conv}}[n] + \sum_{c=1}^{C} \sum_{j=1}^{J} \sum_{k=1}^{K} \boldsymbol{X}^{\text{conv}}[b, c, v + j, u + k].\boldsymbol{\Theta}^{\text{conv}}[n, c, j, k] \tag{1}$$

[1]The computational transforms discussed in sections 3 and approximate computing techniques detailed in section 5 can both be employed during the training and the inference.

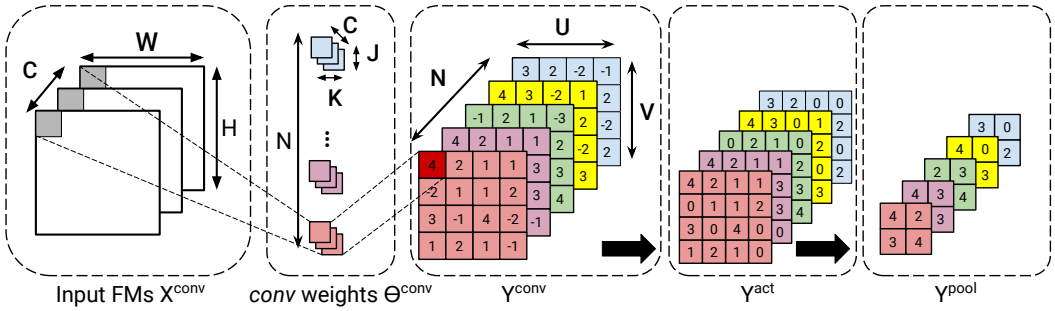


Figure 1: Feed forward propagation in *conv*, *act* and *pool* layers (Batch size B=1, bias β omitted)

2.3.2 Activation Layers:

Each *conv* layer of a CNN is usually followed by an activation layer that applies a *non-linear* function to all the values of FMs. Early CNNs were trained with TanH or Sigmoid functions but recent models employ the Rectified Linear Unit (ReLU) function that grants faster training times and less computational complexity, as highlighted in [14].

$$\forall \{b, n, u, v\} \in [1, B] \times [1, N] \times [1, V] \times [1, U]$$
$$Y^{\text{act}}[b, n, h, w] = \text{act}\Big(X^{\text{act}}[b, n, h, w]\Big) \qquad | \qquad \text{act} := \text{TanH, Sigmoid, ReLU ...} \tag{2}$$

2.3.3 Pooling layers:

The convolutional and activation parts of a CNN are directly inspired by the cells of visual cortex in neuroscience [15]. This is also the case of *pooling* layers, which are periodically inserted in-between successive *conv* layers. As shown in equation 3, *pooling* sub-samples each channel of the input FMs by selecting the *average*, or, more commonly, the *maximum* of a given neighborhood K. As a results, the dimensionality of a FMs is reduced, as illustrated in figure 1

$$\forall \{b, n, u, v\} \in [1, B] \times [1, N] \times [1, V] \times [1, U]$$
$$Y^{\text{pool}}[b, n, v, u] = \max_{p,q \in [1:K]} \Big(X^{\text{pool}}[b, n, v + p, u + q]\Big) \tag{3}$$

2.3.4 Fully Connected Layers:

When deployed for classification tasks, the CNNs pipeline is often terminated by Fully Connected (FC) layers. These layers can be seen as *conv* layers with no weight sharing (i.e $W = K$ and $H = J$). Moreover, in a same way as *conv* layers, a non-linear function is applied to the outputs of *FC* Layers.

$$\forall \{b, n\} \in [1, B] \times [1, N]$$
$$Y^{\text{fc}}[b, n] = \beta^{\text{fc}}[n] + \sum_{c=1}^{C} \sum_{h=1}^{H} \sum_{w=1}^{W} X^{\text{fc}}[b, c, h, w].\Theta^{\text{fc}}[n, c, h, w] \tag{4}$$

2.3.5 Batch-Normalization Layers:

Batch-Normalization is introduced in [16] to speed up training by linearly shifting and scaling the distribution of a given batch of inputs B to have zero mean and unit variance. These layers find also there interest when implementing Binary Neural Network (BNN) (cf section 5.1.3) by reducing the quantization error compared to an arbitrary input distribution, as highlighted in [17]. Equation 5 details the processing of *batch norm* layers, where μ and σ are statistic collected during the training, α , ϵ and γ parameters are training hyper-parameters.

$$\forall \{b, n, u, v\} \in [1, B] \times [1, N] \times [1, V] \times [1, U]$$
$$\boldsymbol{Y}^{\mathrm{BN}}[b, n, v, u] = \frac{\boldsymbol{X}^{\mathrm{BN}}[b, n, u, v] - \mu}{\sqrt{\sigma^2 + \epsilon}} \gamma + \alpha \tag{5}$$

2.4 Workload of a CNNs inference

Table 2: Popular CNN models with their computational workload. Accuracy measured on single-crops of ImageNet test-set.

Model	AlexNet [14]	GoogleNet [18]	VGG16 [6]	VGG19 [6]	ResNet50 [19]	ResNet101 [19]	ResNet-152 [19]
Top1 err	42.9 %	31.3 %	28.1 %	27.3 %	24.7%	23.6% %	23.0%
Top5 err	19.80 %	10.07 %	9.90 %	9.00 %	7.8 %	7.1 %	6.7 %
conv layers	5	57	13	16	53	104	155
conv workload (MACs)	666 M	1.58 G	15.3 G	19.5 G	3.86 G	7.57 G	11.3 G
conv parameters	2.33 M	5.97 M	14.7 M	20 M	23.5 M	42.4 M	58 M
Activation layers	ReLU						
pool layers	3	14	5	5	2	2	2
FC layers	3	1	3	3	1	1	1
FC workload (MACs)	58.6 M	1.02 M	124 M	124 M	2.05 M	2.05 M	2.05 M
FC parametrs	58.6 M	1.02 M	124 M	124 M	2.05 M	2.05 M	2.05 M
Total workload (MACs)	724 M	1.58 G	15.5 G	19.6 G	3.86 G	7.57 G	11.3 G
Total parameters	61 M	6.99 M	138 M	144 M	25.5 M	44.4 M	60 M

The accuracy of CNN models have been increasing since their breakthrough in 2012 [14]. However, this accuracy comes at the price of a high computational cost. The main challenge that faces CNN developers is to improve classification accuracy while maintaining a tolerable computational workload. As shown in table 2, this challenge was successfully addressed by Inception [18] and ResNet models [19], with their use of bottleneck 1×1 convolutions that reduce both model size and computations while increasing depth and accuracy.

2.4.1 Computational Workload:

The computational workload of a CNN inference is the result of an intensive use of the Multiply Accumulate (MAC) operation. Most of these MACs occur on the convolutional parts of the network, as shown in tab 2. As a consequence, *conv* layers are responsible, in a typical implementation, of more than 90% of execution time during the inference [20]. Conversely to computations, and as shown in tab 2, most of the CNN weights are included on the FC-layers. Due to this unbalanced computation to memory ratio, CNNs accelerators follow different strategies when implementing the convolutional and fully connected parts of inference.

2.4.2 Parallelism in CNNs:

Because of the high number of required computations, inferring CNNs with real-time constraints is a challenge, especially on low-energy embedded devices. A solution to this challenge is to take advantage of the extensive concurrency exhibited by CNNs. These sources can be formalized as:

- **Batch Parallelism:** CNN implementations can simultaneously classify multiple frames grouped as a *batch* B in order to reuses the filters in each layer and minimize the external memory accesses. As a result, the inference benefits from a significant acceleration when implementing batch processing.
- **Inter-layer Parallelism:** CNNs have a feed-forward hierarchical structure consisting of a succession of data-dependent layers. These layers can be executed in a pipelined fashion by launching layer (ℓ) before ending the execution of layer ($\ell - 1$).

Moreover, the computation of each *conv* layer, described in eq 1, exhibits four sources of concurrency that are detailed above.

- **Inter-FM Parallelism:** Each output FM plane of a *conv* layer can be processed separately from the others. This means that P_N elements of $\boldsymbol{Y}^{\text{conv}}$ can be computed in parallel ($0 < P_N < N$).
- **Intra-FM Parallelism:** Multiple pixels of a single output FM plane can be processed concurrently by evaluating $P_V \times T_U$ Values of $\boldsymbol{Y}^{\text{conv}}[n]$ ($0 < P_V \times P_U < V \times U$)
- **Inter-convolution Parallelism:** 3D-convolutions occurring in *conv* layers can be expressed as a sum of 2D convolutions as shown in equation 6. These 2D convolutions can be evaluated simultaneously by computing concurrently P_C elements of eq 6 ($0 < P_C < C$).
- **Intra-convolution Parallelism:** The 2D-convolutions involved in the processing of *conv* layers can be implemented in a pipelined fashion as in [21]. In this case $P_J \times P_K$ multiplications are implemented concurrently ($0 < P_J \times P_K < J \times K$).

$$\begin{aligned} &\forall \{b, n\} \in [1, B] \times [1, N] \\ &\boldsymbol{Y}^{\text{conv}}[n] = b[n] + \sum_{c=1}^{C} \mathbf{conv2D}\Big(\boldsymbol{X}^{\text{conv}}[c], \boldsymbol{\Theta}^{\text{conv}}[n, c]\Big) \end{aligned} \tag{6}$$

2.4.3 Memory Accesses in CNNs:

The CNN inference shows large vectorization opportunities that are exploited by allocating multiple computational resources to accelerate the processing. However, this method may be inefficient if no caching strategy is implemented.

In fact, memory bandwidth is often the bottleneck when processing CNNs. For the *FC* parts, execution can be memory-bounded because of the high number of weights that these layers contain, and consequently, the high number of memory reads engendered. For the *conv* parts, the high number of MAC operations results in a high amount of memory accesses because each MAC requires at least 2 memory reads and 1 memory write to be performed[2]. If all these accesses are towards external memory (for instance, Dynamic Random Access

[2]This is the best case scenario of a fully pipelined MAC where intermediate results don't need to be loaded.

Memory (DRAM)), throughput and energy consumption will be highly impacted since a DRAM access engenders significantly more of latency and energy consumption than the computation it self [22]

The number of these DRAM accesses, and thus latency and energy consumption, can be reduced by implementing a memory caching hierarchy using on-chip memories. As discussed in section 4, Hardware accelerators for CNNs usually employ two levels of caches. The first level is implemented by means of large on-chip buffers while the second level involves local register files implemented at the nearest of the computational capabilities. The latency and energy consumption that result from memory access toward these 2 cache levels is several order of magnitude less then external memory access, as pointed-out in [12].

2.4.4 Hardware, libraries and frameworks:

In order to catch the parallelism of CNNs, dedicated hardware accelerators are developed. Most of them are based on GPU, which that are known to perform well on regular parallelism patterns thanks to a Single Instruction on Multiple Data (SIMD) and Single Instruction on Multiple Threads (SIMD) execution models, a dense collection of floating-point computing elements that peaks at 12 TFLOPs, and high capacity/bandwidth on/off-chip memories [23]. To support these hardware accelerators, specialized libraries for deep learning are developed to provide the necessary programming abstraction, such as CudNN on Nvidia GPUs [24] and DeepCL on heterogeneous hardware through OpenCL standard[25]. Built-upon these libraries, dedicated frameworks for deep learning are proposed to improve productivity of conceiving, training and deploying CNNs, such as Caffe[26] and TensorFlow [27].

Beside GPU implementations, numerous FPGA accelerators for CNNs have been proposed. FPGAs are fine-grain programmable devices that can catch the CNN parallelism patterns with no memory bottleneck, tanks to

1. A High density of hard-wired Digital Signal Processing (DSP) blocs that are able to achieve up to 20 (8 TFLOPs) TMACs [10].
2. A collection of *In-situ* on-chip memories, located next to DSPs, that can be exploited to significantly reduce the number of external memory accesses.

When porting a CNN to an FPGA device, the problem boils down to finding an efficient mapping between the computational model of the former and the execution model supported by the latter. In the the following sections, the main strategies explored by the literature to address this mapping problem are reviewed. In particular, we show that current FPGA-based accelerators for CNNs rely on one (or a combination) of three main optimizations to efficiently infer CNNs.

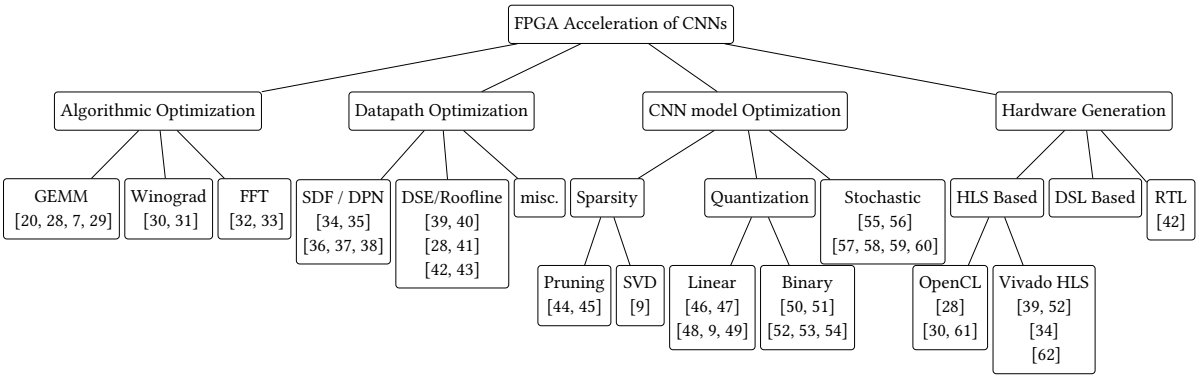


Figure 2: Main Approaches to Accelerate CNN inference on FPGAs

3 Algorithmic Optimizations for FPGA-Based CNN Acceleration

In order to accelerate the execution of *conv* and FC layers, computational transforms are employed on the FMs and kernels in order to vectorize the implementations and reduce the number of arithmetic operations occurring during inference. These computational transforms are mainly deployed in CPUs and GPU and are implemented by means of variety of software libraries such OpenBlas CPUs and cuBLAS for GPUs. Beside this, various implementations make use of such transforms to map CNNs on FPGAs.

3.1 GEMM Transformation

In Central Processing Units (CPUs) and GPUs, a common way to process CNNs is to map *conv* and FC layers as General Matrix Multiplications (GEMMs). The OpenCL standard generalizes this approach to FPGAs-based implementations [63, 64].

For FC layers, in which the processing boils down to a matrix-vector multiplication problem, the GEMM-based implementations find its interest when processing a *batch* of FMs. In this case, the batch is concatenated onto a $CHW \times B$ matrix, as shown in fig 3a.

As mentioned in section 2.4.1, most of the weights of CNNs are employed in the FC parts. Instead of loading these weights multiple times to classify multiple inputs, feature maps of FC layers are *batched* in a way that FC weights are loaded only one time per batch. This vectorization is employed in [65, 66, 30] to increase the computational throughput in FC layers while maintaining a constant memory bandwidth utilization. Moreover, the efficiency of this method increases as the sparsity of Θ^{fc} grows (cf. sec 5.2).

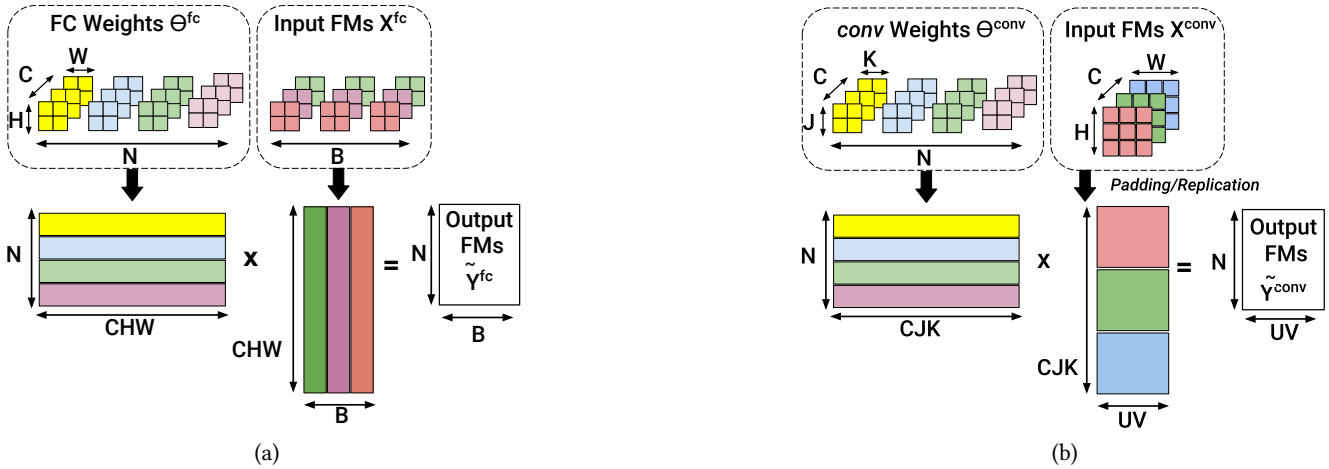


Figure 3: GEMM Based processing of: a- FC layers, b- conv layers.

3D Convolutions can also be mapped as GEMMs using, for instance, the computational transform introduced in [29]. Suda et *al.* [28] and more recently, Zhang et *al.* [61] leverage on this GEMMs transcription of 3D convolution to derive OpenCL-based FPGA Accelerators for CNNs. In these works, a transformation flattens all the filters of a given *conv* layer onto an $N \times CKJ$ matrix $\tilde{\Theta}$ and re-arranges input FMs onto a $CKJ \times UV$ matrix $\tilde{X}$. The output FMs, $\tilde{Y}$, is the result of the multiplication of the two former matrices, as illustrated in Fig 3b. The mapping of *conv* layers as GEMMs can also be performed using a relaxed form of the Toeplitz matrix [67]. However, the downside for using GEMMs for the layers is the introduction of redundant data in the input FMs. This redundancy, as pointed-out in [12], can lead to either inefficiency in storage or complex memory access patterns. As a result, other strategies to map convolutions are considered.

$$\tilde{Y}^{\text{conv}} = \tilde{\Theta}^{\text{conv}} \times \tilde{X}^{\text{conv}} \tag{7}$$

3.2 Winograd Transform

Winograd minimal filter algorithm, introduced in [68], is a computational transform that can be applied to convolutions when the stride is 1. Winograd convolutions are particularly efficient when processing small convolutions ($K \leq 3$), as demonstrated in [69]. In this works, authors report an acceleration up to x7.28 when compared to classical GEMM based implementation of convolutions when executing VGG16 on a TitanX GPU.

In Winograd filtering, data is processed by blocs referred as *tiles*, as following:

1. An input FM tile x of size $(u \times u)$ is pre-processed: $\tilde{x} = \boldsymbol{A}^T x \boldsymbol{A}$
2. In a same way, θ the filter tile of size $(k \times k)$ is transformed into $\tilde{\theta}$: $\tilde{\theta} = \boldsymbol{B}^T x \boldsymbol{B}$
3. Winograd filtering algorithm, denoted $F(u \times u, k \times k)$, outputs a tile y of size $(u \times u)$ that is computed according to equation 8

$$y = \boldsymbol{C}^T \left[\tilde{\theta} \odot \tilde{x} \right] \boldsymbol{C} \tag{8}$$

where $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ are transformation matrices defined in the Winograd algorithm [68] and $\odot$ denotes the Hadamard product or Element-Wise Matrix Multiplication (EWMM).

While a standard filtering requires $u^2 \times k^2$ multiplications, Winograd algorithm $F(u \times u, k \times k)$ requires $(u+k-1)^2$ multiplications [68]. In the case of tiles of a size $u = 2$ and kernels of size $k = 3$, this corresponds to an arithmetic complexity reduction of x2.25 [69]. In return, the number of additions is increased.

Beside this complexity reduction, implementing Winograd filtering in FPGA-Based CNN accelerators has two advantages. First, transformation matrices $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ can be generated off-line once u and k are determined. As a result, these transforms become multiplications with the constants that can be implemented by means of Lookup Table (LUT) and shift registers, as proposed in [70].

Second, Winograd filtering can employ the loop optimization techniques discussed in section 4.2 to vectorize the implementation. On one hand, the computational throughput is increased when *unrolling* the computation of the EWMMs parts on an array of DSP blocs. On the other hand, memory bandwidth is optimized using loop *tiling* to determine the size FM tiles and filter buffers.

First utilization of Winograd filtering in FPGA-Based CNN accelerators is proposed in [31] and delivers a computational throughput of 46 GOPs when executing AlexNet convolution layers. This performance is significantly by a factor of x42 in [30] when optimizing the datapath to support Winograd convolutions (by employing loop unrolling and tiling strategies), and storing the intermediate FM in on-chip buffers (cf sec 4). The same methodology is employed in [70] to derive a CNN accelerator on a Xilinx ZCU102 device. This accelerator delivers a throughput of 2.94 TOPs on VGG convolutional layers, which corresponds to half of the performance of a TitanX device, with x5.7 less power consumption [23][3].

3.3 Fast Fourier Transform

Fast Fourier Transofrm (FFT) is a well known algorithm to transform the 2D convolutions into EWMM in the frequency domain, as shown in equation 9:

$$\textbf{conv2D}(X[c], \Theta[n, c]) = \text{IFFT}\Big(\text{FFT}(X[c]) \odot \text{FFT}(\Theta[n, c])\Big) \tag{9}$$

Using FFT to process 2D convolutions reduces the arithmetic complexity to $O(W^2 log_2(W))$, which is exploited to derive FPGA-based accelerators to *train* CNNs [33]. When compared to standard filtering and Winograd algorithm, FFT finds its interest in convolutions with large kernel size ($K > 5$), as demonstrated in [69, 63]. The computational complexity of FFT convolutions can be further reduced to $O(W log_2(K))$ using the Overlap-and-Add Method [71] that can be applied when the signal size is much larger than the filter size, which is the case in

[3]Implementation in the TitanX GPU employs Winograd algorithm and 32 bits floating point arithmetic

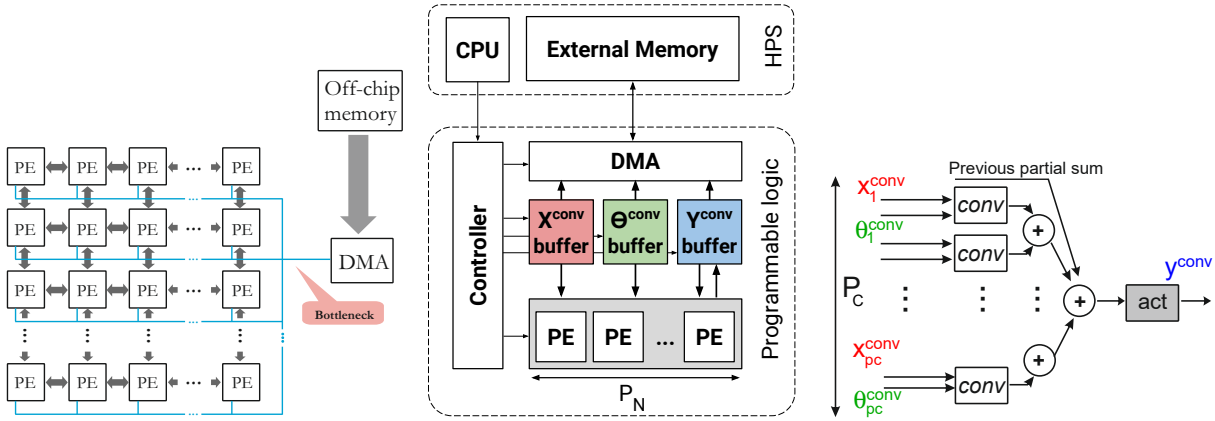
Table 3: FPGA-Based CNN accelerators employing computational transform to accelerate conv layers

		Network	Network Workload		Bitwidth	Desc.	Device	Freq	Through	Power	LUT	DSP	Memory
			Comp. (GOP)	Param. (M)				(MHz)	(GOPs)	(W)	(K)		(MB)
Winograd	[31]	AlexNet-C	1.3	2.3	Float 32	OpenCL	Virtex7 VX690T	200	46		505	3683	56.3
	[30]	AlexNet-C	1.3	2.3	Float16	OpenCL	Arria10 GX1150	303	1382	44.3	246	1576	49.7
	[70]	VGG16-C	30.7	14.7	Fixed 16	HLS	Zynq ZU9EG	200	3045	23.6	600	2520	32.8
		AlexNet-C	1.3	2.3					855				
FFT	[32]	AlexNet-C	1.3	2.3	Float 32		Stratix5 QPI	200	83	13.2	201	224	4.0
		VGG19-C	30.6	14.7					123				
GEMM	[28]	AlexNet-C	1.3	2.3	Fixed 16	OpenCL	Stratix5 GXA7	194	66	33.9	228	256	37.9
	[66]	VGG16-F	31.1	138.0	Fixed 16	HLS	Kintex KU060	200	365	25.0	150	1058	14.1
							Virtex7 VX960T	150	354	26.0	351	2833	22.5
	[61]	VGG16-F	31.1	138.0	Fixed 16	OpenCL	Arria10 GX1150	370	866	41.7	437	1320	25.0
					Float 32	OpenCL		385	1790	37.5		2756	29.0

conv layers ($W >> K$). Works in [32] exploit this method to implement frequency domain acceleration for *conv* layers on FPGA, which results in a computational throughput of 83 GOPs for AlexNet.

4 Data-path Optimizations for FPGA-Based CNN Accelerators

As highlighted in sec 2.4.2, the execution of CNNs exhibit numerous sources of parallelism. However, because of the resource limitation of FPGAs devices, it is impossible to fully exploit all the parallelism patterns, especially with the sheer volume of operations involved in deep topologies. In other words, the execution of recent CNN models can not fully be "Unrolled", sometimes, not even for a single *conv* layer. To address this problem, the main approach that state-of-the-art implementations advocates, is to map a limited number of Processing Elements (PEs) on the FPGA. These PEs are reused by temporally iterating data through them.



(a) Static Systolic Array (b) Generic SIMD Accelerator (c) Processing Element

Figure 4: Generic Data-paths of FPGA-based CNN accelerators

4.1 Systolic Arrays

Early FPGA-based accelerators for CNNs implemented systolic arrays to accelerate the 2D filtering in convolutions layers [72, 73, 74, 75, 76]. As illustrated in figure 4a, systolic arrays employ *a static collection* of PEs, typically

Table 4: Loop Optimization Parameters P_i and T_i

Parallelism	Intra-layer	Inter-FM	Intra-FM		Inter-Convolution	Intra-Convolution	
Loop	L_L	L_N	L_V	L_U	L_C	L_J	L_K
Unroll factor	P_L	P_N	P_V	P_U	P_C	P_J	P_K
Tiling Factor	T_L	T_N	T_U	T_U	T_C	T_J	T_K

arranged in a 2-dimensional grid, that operates under the control of a CPU. This static collection of PEs is agnostic to the CNN model configuration. It can only support convolutions with a kernel size K that is smaller than a given maximum size K_m (i.e support only convolutions such $K \leq K_m$ where , for instance, $K_m = 7$ in [73] and $K_m = 10$ in [76]). Moreover, when performing convolutions with a smaller kernel size then K_m ($K << K_m$), only a small part of computing capabilities is used. For instance in [76], processing 3×3 convolutions uses only 9% of DSP Blocs. Finally, these systolic arrays do not implement data caching and requires to fetch inputs from off-chip memory. As a result, their performance is bounded by memory bandwidth of the device.

4.2 SIMD Accelerators and Loop Optimization

Due to inefficiency of static systolic arrays, flexible SIMD accelerators for CNNs on FPGAs were proposed. The general computation flow in these accelerators –illustrated in Fig.4c-a– is to fetch FMs and weights from DRAM to on-chip buffers. These data are then streamed into the PEs. At the end of the PE computation, results are transferred back to on-chip buffers and, if necessary, to the external memory in order to be fetched in their turn to process the next layers. Each PE –as depicted in Fig. 4c-b– is configurable and has its own *computational* capabilities by means of DSP blocs, and its own data *caching* capabilities by means of on-chip registers.

With this paradigm, the problem of CNN mapping boils down to finding the optimal architectural configuration of PEs (number of PEs, number of DSP blocs per PE, size of data caches), as well as the optimal temporal scheduling of data that maximizes the computational throughput $\mathcal{T}$.

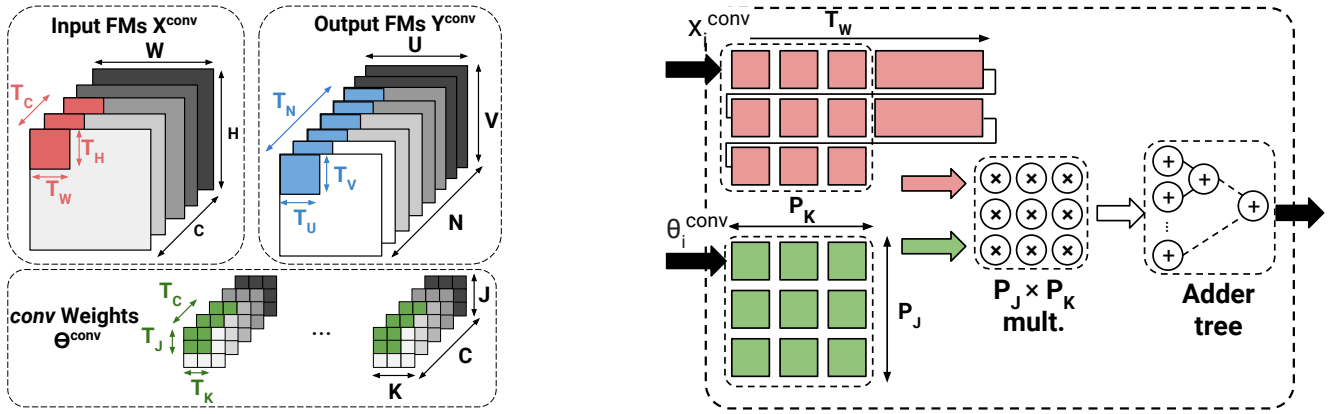
For convolution layers, in which the processing is described in listing 6a, finding the optimal PE configuration can be seen as a loop optimization problem [39, 9, 28] [77, 65, 40, 78, 36, 79, 80, 43]. This problem is addressed by applying loop optimization techniques such *loop unrolling*, *loop tiling* or *loop interchange* to the 7 nested loops of listing 6a. In this case, setting the unroll and tiling factors (*resp.* P_i and T_i) determines the number of PEs, the computational resources and on-chip memory allocated to each PE in addition to the size of on-chip buffer and the amount of DRAM accesses.

4.2.1 Loop Unrolling:

Unrolling a loop L_i with an unrolling factor P_i ($P_i \leq i, i \in \{L, V, U, N, C, J, K\}$) accelerates its execution at the expense of resource utilization. Each of the parallelism patterns listed in section 2.4.2 can be implemented by unrolling one of the loops of listing 6a, as summarized in table 4. For configuration given in figure 4c, the unrolling factor P_N determines the number of PEs. On the other hand, unrolling factors P_C, P_K, P_J determine the number of multipliers and adders, as well as the size of registers contained in each PE.

4.2.2 Loop Tiling:

In general, the capacity of on-chip memory in current FPGAs is not large enough to store all the weights and intermediate FMs of all CNN layersAs a consequence, FPGA based accelerators resort to external DRAMs to store this data. As mentioned in section 2.4.3, DRAM accesses are costly in terms of energy and latency, and data caches



(a) (b)

Figure 5: Loop tiling and unrolling

must be implemented by means of on-chip buffers and local registers. The challenge is to configure the data-path in a way that every data transferred from DRAM is reused as much as possible.

For *conv* layers, this challenge can be addressed by *tiling* the nested loops of listing 6a. *Loop tiling* [81] divides the FMs and weights of each layer into multiple blocks that can fit into the on-chip buffers. For the configuration given in figure 4c, sizes of buffers containing input FM, weights and output FM are determined by the tiling factors detailed in table 4, according to equation 10

$$\mathcal{M}_{\text{conv}} = T_C T_H T_W + T_N T_C T_J T_K + T_N T_V T_U \tag{10}$$

```
// Ll: Layer
for (int l=0;l<L,l++){
// Lb : Batch
for (int b=0;b<B,l++){
// Ln: Y Depth
for (int n=0;n<N;n++){
// Lv: Y Columns
for (int v=0;v<V,v++){
// Lu: Y Raws
for (int u=0;u<U,u++){
// Lc: X Depth
for (int c=0;n<C;c++){
// Lj: Theta Columns
for (int j=0;j<J,j++){
// Lk: Theta Raws
for (int k=0;k<K,k++){
  Y[b,l,n,v,u] += X[b,l,c,v+j,u+k] *
      Theta[l,n,c,j,k]
}}}}}}}
```

(a)

```
for (int n=0;n<N;n+=Tn){
for (int v=0;v<V,v+=Tv){
for (int u=0;u<U,u+=Tu){
for (int c=0;n<C;c+=Tc){
  // DRAM: Load in on-chip buffers the tiles:
  // X[l,c:c+Tc,v:v+Tv,u:u+Tu]
  // Theta [l,n:n+Tn,c:c+Tc,j,k]
  // Process on-chip tiles
  for (int tn=0;tn<Tn;tn++){
  for (int tv=0;tv<Tv,tv++){
  for (int tu=0;tu<Tu,tu++){
  for (int tc=0;tn<Tc;tc++){
  for (int j=0;j<J,j++){
  for (int k=0;k<K,k++){
    Y[l,tn,tv,tu] += X[l,tc,tv+j,tu+k] *
        Theta[l,tn,tc,j,k];
  }}}}}}
  // DRAM: Store output tile
}}}}
```

(b)

Figure 6: Loop Tiling in conv layers: a-Before tiling, b-After tiling

4.2.3 Design Space Exploration:

In order to find the optimal unrolling and tiling factors, a large exploration of the design space is needed. In a general way, an analytical model is built. Inputs of this model are loop factors P_i, T_i and outputs are a theoretical prediction of the allocated resources, the computational throughout and the memory bandwidth used. This model is parametrized by the available resources of a given FPGA platform and the workload of the CNN.

Given this model, the objective is to find the design parameters that minimize the memory access while maximizing the resource utilization. To address this optimization problem, a brute force exploration is performed, such in [39, 28, 77, 65, 40, 78]. This exploration is usually driven by the Roofline method [82] in order to select the feasible design solutions that matches with the maximum computational throughput and the maximum memory bandwidth a given platform can deliver [39, 40, 41]. The design space can also be explored by means of heuristic search algorithms, as proposed for instance in [35].

4.2.4 FPGA Implementations:

Employing loop optimizations to derive FPGA-based CNN accelerator was first investigated in [39]. In this work, Zhang *et al.* report a computational throughput of 61.62 GOPs in the execution of AlexNet convolutional layers by unrolling loops L_C and L_N. This accelerator was built using HLS tools and rely on 32 floating point arithmetic. Works in [78] follow the same unrolling scheme and implement the FC part of the inference. Moreover, design [78] features 16 bits fixed point arithmetic and RTL conception, resulting in a x2.2 improvement in terms of computational throughput. Finally, the same unrolling and tiling scheme are employed in recent works [65] were authors report a x13,4 improvement over their original works in [39], thanks to a deeply pipelined FPGA cluster of four Virtex7-XV960t devices and a 16 bits fixed point arithmetic.

In all these implementations, loops L_J and L_K are not unrolled because J and K are usually small, especially in recent topologies (cf Table 2). Works of Motamedi *et al.* [40] study the impact of unrolling these loops in AlexNet, where the first convolution layers use 11×11 and 5×5 filters. Expanding loop unrolling and tiling to loops L_J and L_K results in a x1.36 improvement in computational throughput *vs* [39] on the same VX485T device when using 32 floating point arithmetic. In a same way, implementations in [28, 9, 36] tile and unroll loops L_N, L_C, L_J, L_K and demonstrate higher acceleration on AlexNet and VGG when using fixed point arithmetic. Nevertheless, and as pointed out in [80], unrolling loops L_J and L_K is ineffective for recent CNN models that employ small convolution kernels. In addition, Tiling loops L_J and L_K requires PEs to be configured differently for different layers, increasing thus the control complexity.

The values of U, V, N can be very large in CNN models. Consequently, unrolling and tiling loops L_U, L_V, L_N can be efficient only for devices with high computational capabilities (i.e DSP Blocs). This is demonstrated in works of Rahman *et al.* [77] that report an improvement of $\times1.22$ over [39] when enlarging the design space exploration to loops L_U, L_V, L_N

In order to keep data in on-chip buffer after the execution of a given layer,[79] investigates fused-layer CNN Accelerators by tiling across layer L_L. As a result, authors report a reduction of 95% of DRAM accesses at the cost of 362KB of extra on-chip memory.

In all these approaches, loops L_N, L_C, L_J, L_K are unrolled in a same way they are tilled (i.e $T_i = P_i$). By contrast, the works of Ma *et al.* [80, 83] fully explore all the design variables searching for optimal loop unroll and tiling factors. More particularly, authors demonstrate that the input FMs and weights are optimally reused when unrolling only computations within a single input FM (i.e when $P_C = P_J = P_k = 1$). Tiling factors are set in way that all the data required to compute an element of Y are fully buffered (i.e $T_C = C, T_K = K, T_J = J$). The remaining design parameters are derived after a brute force design exploration. The same authors leverage on these loop optimizations to build an RTL compiler for CNNs in [84]. To the best of our knowledge, this accelerator

Table 5: FPGA-based CNN accelerators implementing loop optimization

	Network	Network Workload		Bitwidth	Desc.	Device	Freq	Through	Power	LUT	DSP	Memory
		Comp. (GOP)	Param. (M)				(MHz)	(GOPs)	(W)	(K)		(MB)
[39]	AlexNet-C	1.3	2.3	Float 32	HLS	Virtex7 VX485T	100	61.62	18.61	186	2240	18.4
[9]	VGG16SVD-F	30.8	50.2	Fixed 16	HDL	Zynq Z7045	150	136.97	9.63	183	780	17.5
[28]	AlexNet-C	1.3	2.3	Fixed 16	OpenCL	Stratix5 GSD8	120	187.24	33.93	138	635	18.2
	AlexNet-F	1.4	61.0					71.64		272	752	30.1
	VGG16-F	31.1	138.0					117.9		524	1963	51.4
[77]	AlexNet-C	1.3	2.3	Float 32	HLS	Virtex7 VX485T	100	75.16		28	2695	19.5
[65]	AlexNet-F	1.4	61.0	Fixed 16	HLS	Virtex7 VX690T	150	825.6	126.00		14400	
	VGG16-F	31.1	138.0					1280.3	160.00		21600	
[78]	NIN-F	2.2	61.0	Fixed 16	HDL	Stratix5 GXA7	100	114.5	19.50	224	256	46.6
	AlexNet-F	1.5	7.6					134.1	19.10	242	256	31.0
[36]	AlexNet-F	1.4	61.0	Fixed 16		Virtex7 VX690T	156	565.94	30.20	274	2144	34.8
[79]	AlexNet-C	1.3	2.3	Float 32	HLS	Virtex7 VX690T	100	61.62		273	2401	20.2
[80]	VGG16-F	31.1	138.0	Fixed 16	HDL	Arria10 GX1150	150	645.25	50.00	322	1518	38.0
[43]	AlexNet-F	1.4	61.0	Fixed 16	OpenCL	Arria10 GT1150	239.6	360.4		700	1290	47.2
	VGG-F	31.1	138.0				221.65	460.5		708	1340	49.3
	VGG-F	31.1	138.0				231.85	1171.3		626	1500	33.4
[42]	AlexNet-C	1.3	2.3	Fixed 16	HDL	Cyclone5 SEM	100	12.11		22	28	0.2
		1.3	2.3			Virtex7 VX485T	100	445			2800	
[84]	NiN	20.2	7.6	Fixed 16	HDL	Stratix5 GXA7	150	282.67		453	256	30.2
	VGG16-F	31.1	138.0					352.24		424	256	44.0
	ResNet-50	7.8	25.5					250.75		347	256	39.3
	NiN	20.2	7.6			Arria10 GX1150	200	587.63		320	1518	30.4
	VGG16-F	31.1	138.0					720.15		263	1518	44.5
	ResNet-50	7.8	25.5					619.13		437	1518	38.5
[85]	AlexNet-F	1.5	7.6	Float 32		Virtex7 VX690T	100	445.6	24.80	207	2872	37
	VGG16SVD-F	30.8	50.2					473.4	25.60	224	2950	47

outperforms all the previous implementations that are based on loop optimization in terms of computational throughput.

4.3 Dataflow MoC For CNNs

Feed-forward propagation is by nature a streaming based applications in which the execution is purely data-driven. In fact, the CNN layout is in contrast with Von Neumann execution models and a CNN implementation can easily be memory-bounded if it has to fetch every instruction from memory. This motivated multiple approaches to investigate the applicability of the data-flow Model of Computation (MoC) to accelerate CNNs on FPGAs.

The foundations of the data-flow MoC were formalized by [86] in order to create an architecture where multiple fragments of instructions can process simultaneously streams of data. Programs respecting dataflow semantics are described as Data-flow Process Networks (DPNs). Each node of this network corresponds to a fundamental processing unit called an *actor* and each edge corresponds to a communication *FIFO* channel. Actors exchange abstract data –known as *tokens*– through these FIFOs. Each actor follows a purely data-driven execution model wherein the *firing* (execution) is triggered only by the availability of input operands. This is typically the case in CNNs, where the execution of each layer is only triggered by the availability of input FM.

Applying the data-flow MoC to accelerate CNN implementations on FPGAs is investigated in [87]. In this work, authors demonstrate the efficiency of the proposed *lightweight data-flow methodology* [88] by mapping

Table 6: FPGA-Based CNN accelerators employing the data-flow MoC

	Network	Network Workload		Bitwidth	Desc.	Device	Freq	Through	Power	LUT	DSP	Memory
		Comp. (GOP)	Param. (M)				(MHz)	(GOPs)	(W)	(K)		(KB)
[91]	CarType-C	0.16	0.03	Float 32	HDL	Zynq Z7045	100	0.47	0.23	68	24	1440.0
[34]	LeNet5-C	0.04	0.03	Fixed 16	HLS	Zynq Z7020	100	0.48	0.75	14	4	42.7
	SignRecog-C	4.03	0.04					6.03		26	144	38.2
[90]	VGG16-F	31.10	138.00	Fixed 16	HLS	Zynq Z7045	125	123.12		219	900	2400.0
[38]	SVHN-C	0.02	0.08	Fixed 5	HDL	Cyclone5 GX	63.96	170.73		40	0	10.9
	LeNet5-C	0.04	0.03	Fixed 3				2438.46		8	0	0.2

conv layers with variable clock-domains in a Zynq ZC706 device.

A special case of data-flow, referred as Static Data-Flow (SDF) [89], is a paradigm in which the number of tokens produced and consumed by each actor can be specified a priori, as it is the case in the CNN execution. SDF model is employed in [34, 90] to optimize the mapping of CNN graphs on FPGAs. In this works, the CNN graph is modeled as a topology matrix that contains the the number of incoming streams, the size of tokens and the consumption rates of each actor. Instead of exploring the design space of unrolling and tiling parameters (cf. sec 4.2), authors explore the design space of the topology matrix components. These optimal components are used to derive the configuration of the PE and buffers that either minimizes the computation latency or energy consumption. Moreover, and in contrast with classical implementations where data is streamed in and out of layers using off-chip data transfers, authors exploit partial dynamic reconfiguration of FPGAs to process different layers.

Finally, works in [38] optimize the direct hardware mapping of CNN graphs. In this approach, each actor of the DPN is physically mapped on the device with its own specific instance, while each edge is mapped as a signal. As all the computations are unrolled, applicability of this method can rapidly be limited by the resource of the device or the size of the CNN, preventing this approach from implementing deep models.

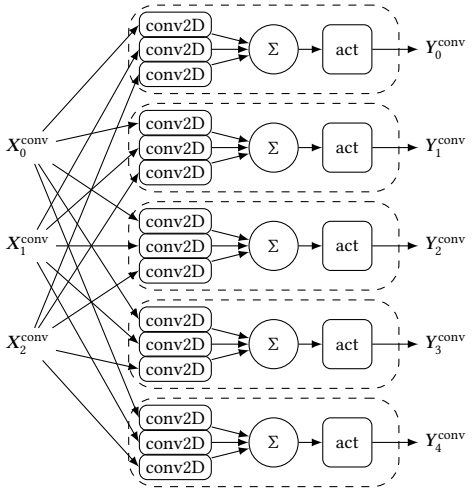


Figure 7: An example of a graph representation of a convolution layer ($C = 3, N = 5$)

5 Approximate Computing of CNN Models

Beside the computational transforms and data-path optimizations, the CNN execution can be accelerated when employing approximate computing which is known to perform efficiently on FPGAs [92].

In this approach, a minimal amount of the CNN accuracy is traded to improve the computational throughput and energy efficiency of the execution. Two main strategies are employed. This first implements approximate *arithmetic* to process the CNN layers with a reduced precision while the second aims to reduce the number of operations occurring in CNN models without critically affecting the modeling performance. Both of these methods can be integrated in the *learning* phase to jointly maximize the accuracy and minimize the workload of a given CNN model.

5.1 Approximate Arithmetic for CNNs

Several studies have demonstrated that the precision of both operations and operands in CNNs[4] can be reduced without critically affecting their predictive performance. This reduction can be achieved by *quantizing* either or both of the CNN inputs, weights and/or FMs using a fixed point numerical representation and implementing *approximate multipliers and adders.*

5.1.1 Fixed point arithmetic:

In a general way, CNN models are deployed in CPUs and GPUs using the same numerical precision they were trained with, relying on *simple-precision floating point* representation. This format employs 32 bits, arranged according to the IEEE754 standard. In FPGAs, implementations such [39, 79, 77] employ this data representation.

Nonetheless, several studies in [93, 46, 94] demonstrate that inference of CNNs can be achieved with a reduced precision of operands. In addition, works in [48, 95, 96, 97] demonstrate the applicability of fixed-point arithmetic to train CNNs. In both cases, FMs and/or weights are *quantized* using a *fixed point representation* scheme. In simplest version of this format, numbers are encoded with the same bit-width (bw) that is set according to the numerical range and the desired precision. More particularly, all the operands share the same exponent (i.e scale factor) that can be seen as as the position of the radix point. In this paper, we refer to this representation as Static Fixed Point (SFP).

When compared to floating point, SFP computing with compact bit-width is known to be more efficient in terms of hardware utilization and power consumption. This is especially true in FPGAs [98], where a single DSP block can either implement *one* 32bits floating point multiplication, *two* 18×19 bits multiplications, or *three* 18×19 multiplications [10].

This motivated early implementations to employ SFP in building FPGA-Based CNN accelerators, such in [72, 73, 74], or in [75, 76], where authors use a 16 bits (Q8.8) format to represent FMs and weights. To prevent overflow, the bit-width is expanded when computing the weighted-sums of convolutions and inner-products. If b_X bits are used to quantize the FM and b_Θ bits are used to quantize the weights, an accumulator of size b_{acc} is used, according to equation 11, which corresponds to accumulators of 48 bits in [73, 74].

$$b_{acc} = b_x + b_\Theta + \max_{\ell \leq L} \left(\log_2 \left(C_\ell K_\ell^2 \right) \right) \tag{11}$$

5.1.2 Dynamic Fixed Point for CNNs:

In deep topologies, it can be observed that distinct parts of a network can have a significantly different numerical range of data. More particularly, the FMs of deep layers tend to have larger numerical range than first FMs, while the weights are generally much smaller than the FMs. As a consequence, the bit-width is expanded to keep the same precision while preventing overflow, as in[74]. As a result, and as pointed-out [48], SFP with its unique shared fixed exponent, is ill-suited to deep learning.

[4] and more generally in neural networks

To address this problem, works in [48, 49] advocates the use of Dynamic Fixed Point (DFP) [99][5]. In DFP, different scaling factors are used to process different parts of the network. More particularly, weights, weighted sums and outputs of each layer are assigned distinct scale factors. The optimal scale factors and bit-widths (i.e the ones that deliver the best trade-off between accuracy loss and computational load) for each layer can be derived after a brute force exploration using dedicated frameworks that supports DFP such [49, 100] for Caffe and [96] for TensorFlow. In addition, these tools can *fine-tune* the CNN model to improve the accuracy of the quantized network.

The FPGA-Based CNN Accelerator proposed in [28] is build upon this quantification scheme and employs different bit-widths to represent the FM, the convolution kernels and the FC weights with *resp.* 16,8,10 bits. Without fine-tuning, authors report a drop of 1% in classification accuracy of AlexNet. For the same network, works of [78] employs 10 bits for FMs, 8 bits for both *conv* and *FC* weights and report an accuracy drop of 0.4%. In a same way, Qiu *et al.* employ DFP to quantize the VGG with 8,8 and 4 bits while reporting 2% of accuracy drop. In these accelerators, dynamic quantization is supported by means of data shift modules [9]. Finally, the accelerator in [42] rely on the Ristretto framework [49] to derive an AlexNet model wherein the data is quantized in 16 bits with distinct scale factors per layer[6].

5.1.3 Extreme quantification with Binary and pseudo-Binary Nets:

Beside fixed point quantification, training and inferring CNNs with *extremely compact data representations*, is a research area that is gaining interest. In particular, works in BinaryConnect [50] investigate the applicability of binary weights (i.e weights with either a value of $-\theta$ or θ) to train CNNs, which lowers both bandwidth requirements and accuracy on ImageNet by respectively 3200% and 19.2% (vs AlexNet Float32 Model). The same authors go further by implementing BNNs [17], with a 1bit representation for both FM and weights. In these networks, negative data is represented as 0 while positive values are represented as 1. As a consequence, the computation of MACs boils down to an XNOR operation followed by a pop-count, as shown in figure 8b. Moreover, Batch normalization is performed before applying of the *sign* activation function in order to reduce the information lost during binarization, as shown in figure 8a. However, a classification accuracy drop of 29.8% is observed on ImageNet when using BNNs. In an attempt to lower the accuracy drop of BNNs, Rastegari *et al.* proposed XNOR-Nets [51] which use different scale factors for binary weights (i.e $-\theta_1$ or $+\theta_2$). Moreover, *Pseudo-Binary Networks*, such DoReFa-Net [101] and QNNs [102] reduce the accuracy drop to 6.5% by employing a slightly expanded bit-width (2 bits) to represent the intermediate FMs. Finally, in Trained Ternary Quantization (TTQ) [103], weights are constrained to three values $-\theta_1, 0, -\theta_2$ (2 bits), but FM are represented in a 32bits float scheme. As a consequence, the efficiency gain of TTQ is not as high as in BNNs.But in turn, TTQ achieves comparable accuracy on ImageNet, within 0.7% of full-precision.

In FPGAs, BNNs benefit from a significant acceleration as the processing of "binary" convolutions can be mapped on XNOR gates followed by a pop count operation, as depicted in figure 8b. Furthermore, and as suggested in [7], pop count operation can be implemented using lookup tables in a way that convolutions are processed only with logical elements. The DSPs blocs are can thus be used to process the batch norm calculation (eq 5, which can be formulated as a linear transform reduces in order reduce the number of operations. This approach is followed in the implementation of [104] to derive an FPGA-Based accelerator for BNNs that achieves 207.8 GOP/s while only consuming 4.7 W and 3 DSP Blocs to classify the Cifar10 dataset. For the same task, works in [52, 105] use a smaller network configuration[7] and reaches a throughput of 2.4 TOP/s when using a larger Zynq 7Z045 Device with 11W Power consumption. For ImageNet classification, Binary Net implementation of [106] delivers an overall throughput 1.9 TOP/s on a Stratix V GSD device. In all these works, the first layer is not binerized

[5] An other approach to address this problem is to use half-precision 16 bits floating point, as used in [30]

[6] Since the same PEs are reused to process different layers, the same bit-width is used with a variable radix point for each layer

[7] The network topology used in this work involves 90% less computations and achieves 7% less classification accuracy on Cifar10

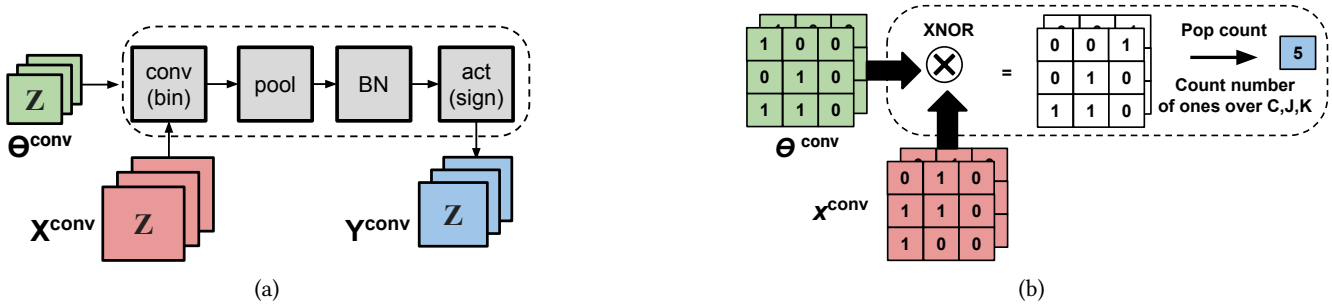


Figure 8: Binary Neural Networks: a-Processing Pipeline, b-Binary Convolutions

to achieve better classification accuracy. As pointed-out in [106], the performance in this layer can be improved when using a higher amount of DSP blocs. Finally, an accelerator for ternary neural networks is proposed in [107] and achieves a peak performance of 8.36 TMAC/s at 13W power consumption for Cifar10 Classification.

5.1.4 Stochastic Computing:

Stochastic Computing (SC) is a low-cost design technique that has been successfully applied in numerous image processing algorithms [108].

In SC, numbers are represented as a random sequence of s bits. In the basic "unipolar" format, the number of ones appearing in the sequence s determines the value of x, i.e the numerical value of a given number x is s_1/s, where x is the number of *ones* appearing in s. The advantage of stochastic arithmetic is that operations are performed with an ultra-small circuitry. For instance, a single AND gate can map a multiplication. Works in [60, 59, 58] demonstrate the feasibility of stochastic arithmetic to accelerate CNNs. More particularly, Ardakani *et al.* propose an FPGA accelerator to classify the MNIST dataset, where multiplications are processed only using AND gates and activation functions (TanH) are implemented in the stochastic domain using FSMs. Such an implementation delivers a computational throughput of 15.44 TOP/s with a misclassification rate of 2.40% on MNIST. However, one the of weakness of SC are long bit-stream. In fact, to represent an n bits number, a bit-stream s of 2^n is required. As a result, stochastic arithmetic suffers from long run-times to perform operations. Moreover, the generation of this bit-streams resorts to dedicated circuitry known as Stochastic Number Generators (SNGs), which add more overhead to the implementation. As a result, SC-based accelerators implement only shallow neural networks with a limited depth.

5.2 Reduce Computations in CNNs

In addition to approximate arithmetic, several studies attempt to the reduce the number of operations involved in CNNs. For FPGA-Based implementation, two main strategies are investigated: *weight pruning*, which increases the *sparsity* of the model, and *low-rank approximation* of filters, which reduces the number of multiplications occurring in the inference.

5.2.1 Weight Pruning:

As highlighted in [109], CNNs as over-parametrized networks and a large amount of the weights can be removed –or *pruned*– without critically affecting the classification accuracy. In its simplest form, pruning is performed according to the magnitude such as the lowest values of the weights are truncated to zero [110]. In a more recent approach, weights removal is driven by energy consumption of a given node of the graph, which is 1.74x more

Table 7: FPGA-Based CNN accelerators employing Approximate arithmetic

		Dataset	Network Workload		Bitwidth				Acc	Device	Freq	Through.	Power	LUT	DSP	Memory
			Comp. (GOP)	Param. (M)	In/Out	FMs	W-C	W-FC			(MHz)	(GOPs)	(W)	(K)		(MB)
FP32	[61]	ImageNet	30.8	138.0	32	32	32	32	90.1	Arria10 GX1150	370	866	41.7	437	1320	25.0
FP16	[30]	ImageNet	1.4	61.0	16	16	16	16	79.2	Arria10 GX1150	303	1382	44.3	246	1576	49.7
DFP	[80]	ImageNet	30.8	138.0	16	16	8	8	88.1	Arria10 GX1150	150	645		322	1518	38.0
	[84]	ImageNet	30.8	138.0	16	16	16	16		Arria10 GX1150	200	720		132	1518	44.5
	[61]	ImageNet	30.8	138.0	16	16	16	16		Arria10 GX1150	370	1790		437	2756	29.0
BNN	[104]	Cifar10	1.2	13.4	20	2	1	1	87.7	Zynq Z7020	143	208	4.7	47	3	
	[52]	Cifar10	0.3	5.6	20/16	2	1	1	80.1	Zynq Z7045	200	2465	11.7	83		7.1
	[106]	MNIST	0.0	9.6	8	2	1	1	98.2	Stratix5 GSD8	150	5905	26.2	364	20	44.2
		Cifar10	1.2	13.4	8	8	1	1	86.3			9396		438	20	
		ImageNet	2.3	87.1	8	32	1a	1	66.8			1964		462	384	
TNN	[107]	Cifar10	1.2	13.4	8	2	2	2	89.4	Xilinx7 VX690T	250	10962	13.6	275		39.4
		SVHN	0.3	5.6					97.6			86124	7.1	155		12.2
		GTSRB	0.3	5.6					99.0			86124	6.6	155		12.2

efficient than magnitude-based approaches [111]. In both approaches, pruning is followed by a fine-tuning of the remaining weights in order to improve the classification accuracy. This is for instance the case in [112], where pruning removes respectively 53% and 85% of the weights in AlexNet *conv* and FC layers for less then 0.5% accuracy loss.

5.2.2 Low Rank Approximation:

Another way to reduce the computations occurring in CNNs is to maximize the number of of *separable filters* in CNN models. A 2D-separable filter θ^{sep} has a unitary rank (i.e $\text{rank}(\theta^{\text{sep}}) = 1$), and can be expressed as two successive 1D filters $\theta_{J\times 1}$ and $\theta_{1\times K}$. When expanding this to 3D filters, a separable 3D convolution requires $C + J + K$ multiplications while a standard 3D convolution requires $C \times J \times K$ multiplications.

Nonetheless, only a small proportion of the filters in CNN Models are separable. To increase this proportion, a first approach is to force the convolution kernels to be separable by penalizing *high rank filters* when training the network [113]. Alternatively, and after the training, the weights $\mathbf{\Theta}$ of a given layer can be approximated into a small set of r *low rank filters* that can be implemented as a succession of fully separable filters. In this case, $r \times (C + J + K)$ multiplications are required to process a single 3D-convolution.

For FC layers, in which the processing boils down to a vector-matrix product, low rank approximation can be achieved by employing, for instance, the SVD decomposition of the weight matrix $\tilde{\mathbf{\Theta}}^{\text{fc}}$ (cf. sec 3.1). Finally, and in a same way to pruning, low rank approximation of weights is followed by a fine-tuning in order counterbalance the classification accuracy drop.

5.2.3 FPGA Implementations:

In FPGA Implementations, low rank approximation is applied on FC layer to significantly reduce the number of weight, such as in [9], where authors derive a VGG16-SVD model that achieves 87.96% accuracy on ImageNet with 63% less parameters.

Sparsity in pruned CNNs can be exploited in FPGA implementations by fully unrolling the processing of a given layer, and skipping (i.e not mapping) the multiplications with zero weights. This approach is investigated in [38], but can be infeasible when the resource of a given device doesn't match with computational requirements of a given layer. Instead, sparsity and pruning can be exploited when processing *conv* and *fc* layers as GEMM

Table 8: FPGA-Based CNN accelerators employing pruning and low rank approximation

		Dataset	Network Workload		Removed	Bitwidth	Acc	Device	Freq	Through.	Power	LUT	DSP	Memory
			Comp. (GOP)	Param. (M)	Param. (%)		(%)		(MHz)	(GOPs)	(W)	(K)		(MB)
SVD	[9]	ImageNet	30.5	138.0	63.6	16 Fixed	87.96	Zynq 7Z045	150	137	9.6	183	780	17.5
Pruning	[45]	Cifar10	0.3	132.9	89.3	8 Fixed	91.53	Kintex 7K325T	100	8621	7.0	17	145	15.1
	[7]	ImageNet	1.4	61.0	85.0	32 Float	79.70	Stratix 10	500	12000	141.2			

(c.f 3.1. In this case, the challenge is to determine the optimal format of matrices that maximizes the chance to detect and skip zero computations, such compressed sparse column (CRC) or compressed sparse row (CSR) formats[8]. Based on previous studied related to sparse GEMM implementation on FPGAs in [114], Sze *et al.*[12] advocates the use of the CRC to process CNNs because this format provides a lower memory bandwidth when the output matrix is smaller then the input, which is typically the case in CNNs where $N < CJK$ in Fig 3b.

However, this efficiency of CRC format is only valid for extremely sparse matrices (typically with $\leq 1\%$ of non zeros), while pruned CNN matrices are not that sparse (typically, $\leq 4 - 80\%$ of non zeros). Therefore, works in [7] use a *zero skip scheduler*, which is an on-chip data manager thanks to which zero elements are identified and not scheduled onto the MAC processing. As a result, the number of cycles required to compute the sparse GEMM is reduced, which corresponds to a 4x speedup in cycle count for and 85% sparse AlexNet layers. Finally, authors report to a projected throughput of 12 TOP/s for pruned CNNs in the next Intel Stratix10 FPGAs, which outperforms and the computational throughput of state-of-the-art GPU implementations by 10%.

6 Conclusion

In this paper, a number of methods and tools have been compared that aim at porting Convolutional Neural Networks onto FPGAs. At the network level, approximate computing and datapath optimization methods have been covered while at the neuron level, the optimizations of convolutional and fully connected layers have been detailed and compared. All the different degrees of freedom offered by FPGAs (custom data types, local data streams, dedicated processors, etc.) are exploited by the presented methods. Moreover, algorithmic and datapath optimizations can a should be jointly implemented, resulting in additive hardware performance gains.

CNNs are by nature overparameterized and support particularly well approximate computing techniques such as weight pruning and fixed point computation. Approximate computing already constitutes a key to CNN acceleration over hardware and will certainly continue driving the performance gains in the years to come.

[8]These format represents a matrix by three one-dimensional arrays, that respectively contain nonzero values, row indices and column indices

Bibliography

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[2] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, and others. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[3] Ross Girshick. Fast R-CNN. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '15*, pages 1440–1448, 2015.

[4] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully Convolutional Networks for Semantic Segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '15*, pages 3431–3440, 2015.

[5] Ying Zhang, Mohammad Pezeshki, Philémon Brakel, Saizheng Zhang, Cesar Laurent Yoshua Bengio, and Aaron Courville. Towards end-to-end speech recognition with deep convolutional neural networks. *arXiv preprint*, arXiv:1701, 2017.

[6] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint*, arXiv:1409:1–14, 2014.

[7] Eriko Nurvitadhi, Suchit Subhaschandra, Guy Boudoukh, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason OngGeeHock, Yeong Tat Liew, Krishnan Srivatsan, and Duncan Moss. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks? In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, pages 5–14, 2017.

[8] Kalin Ovtcharov, Olatunji Ruwase, Joo-young Kim, Jeremy Fowers, Karin Strauss, and Eric Chung. Accelerating Deep Convolutional Neural Networks Using Specialized Hardware. *White paper*, pages 3–6, 2 2015.

[9] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '16*, pages 26–35, New York, NY, USA, 2016. ACM.

[10] Intel FPGA. Intel® Stratix® 10 Variable Precision DSP Blocks User Guide. Technical report, Intel FPGA Group, 2017.

[11] Griffin Lacey, Graham W. Taylor, and Shawki Areibi. Deep Learning on FPGAs: Past, Present, and Future. *arXiv e-print*, 2 2016.

[12] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12):2295–2329, 12 2017.

[13] Y LeCun, L Bottou, Y Bengio, and P Haffner. Gradient Based Learning Applied to Document Recognition. In *Proceedings of the IEEE*, volume 86, pages 2278–2324, 1998.

[14] Alex Krizhevsky, Ilya Sutskever, Hinton Geoffrey E., and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems - NIPS'12*, page 1–9, 2012.

[15] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

[16] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In Francis Bach and David Blei, editors, *Proceedings of the International Conference on Machine Learning - ICML '15*, volume 37, pages 448–456, Lille, France, 2015.

[17] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv e-print*, 2 2016.

[18] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '15*, 2015.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '16*, pages 770–778, 6 2016.

[20] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In *Proceedings of the International Conference on Artificial Neural Networks - ICANN '14*, pages 281–290. Springer, 2014.

[21] Richard G Shoup. Parameterized convolution filtering in a field programmable gate array. In *Proceedings of the International Workshop on Field Programmable Logic and Applications on More FPGAs.*, pages 274–280, 1994.

[22] Mark Horowitz. Computing's energy problem (and what we can do about it). In *IEEE International Solid-State Circuits Conference Digest of Technical Papers - ISSCC '14*, pages 10–14. IEEE, 2 2014.

[23] Nvidia. GPU-Based Deep Learning Inference: A Performance and Power Analysis. *White Paper*, 2015.

[24] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *arXiv e-print*, 2014.

[25] Hugh Perkins. Deep CL: OpenCL library to train deep convolutional neural networks, 2017.

[26] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the ACM International Conference on Multimedia*, 2014.

[27] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, Xiaoqiang Zheng, and Google Brain. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation - OSDI '16*, pages 265–284, 2016.

[28] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '16*, pages 16–25, 2016.

[29] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High Performance Convolutional Neural Networks for Document Processing. 10 2006.

[30] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. An OpenCL(TM) Deep Learning Accelerator on Arria 10. In ACM, editor, *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, pages 55–64, Monterey, California, USA, 2017. ACM.

[31] Roberto DiCecco, Griffin Lacey, Jasmina Vasiljevic, Paul Chow, Graham Taylor, and Shawki Areibi. Caffeinated FPGAs: FPGA Framework For Convolutional Neural Networks. In *Proceedings of the International Conference on Field-Programmable Technology - FPT '16*, 2016.

[32] Chi Zhang and Viktor Prasanna. Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, pages 35–44, 2017.

[33] Jong Hwan Ko, Burhan Ahmad Mudassar, Taesik Na, and Saibal Mukhopadhyay. Design of an Energy-Efficient Accelerator for Training of Convolutional Neural Networks using Frequency-Domain Computation. In *Proceedings of the Annual Conference on Design Automation - DAC '17*, 2017.

[34] Stylianos I. Venieris and Christos Savvas Bouganis. FpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs. In *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines - FCCM '16*, pages 40–47, 2016.

[35] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to FPGAs. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture - MICRO '16*, pages 1–12, 2016.

[36] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *Proceedings of the International Conference on Field Programmable Logic and Applications - FPL '16*, pages 1–9. IEEE, 8 2016.

[37] Giuseppe Natale, Marco Bacis, and Marco Domenico Santambrogio. On How to Design Dataflow FPGA-Based Accelerators for Convolutional Neural Networks. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI - ISVLSI' 17*, pages 639–644. IEEE, 7 2017.

[38] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Serot, Cedric Bourrasset, and François Berry. Tactics to Directly Map CNN graphs on Embedded FPGAs. *IEEE Embedded Systems Letters*, pages 1–4, 2017.

[39] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '15*, FPGA, pages 161–170, 2015.

[40] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. Design space exploration of FPGA-based Deep Convolutional Neural Networks. In *Proceedings of the Asia and South Pacific Design Automation Conference - ASPDAC'16*, pages 575–580, 1 2016.

[41] Paolo Meloni, Gianfranco Deriu, Francesco Conti, Igor Loi, Luigi Raffo, and Luca Benini. Curbing the Roofline : a Scalable and Flexible Architecture for CNNs on FPGA. In *Proceedings of the ACM International Conference on Computing Frontiers - CF '16*, pages 376–383, Como, Italy, 2016.

[42] Mohammad Motamedi, Philipp Gysel, and Soheil Ghiasi. PLACID: A Platform for FPGA-Based Accelerator Creation for DCNNs. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 13(4):62:1–62:21, 9 2017.

[43] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In *Proceedings of the Annual Conference on Design Automation - DAC '17*, pages 1–6, New York, New York, USA, 2017. ACM.

[44] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning Convolutional Neural Networks for Resource Efficient Learning. *arXiv preprint*, 2017.

[45] Tomoya Fujii, Simpei Sato, Hiroki Nakahara, and Masato Motomura. An FPGA Realization of a Deep Convolutional Neural Network Using a Threshold Neuron Pruning. In *Proceedings of the International Symposium on Applied Reconfigurable Computing - ARC'16*, volume 9625, pages 268–280, 2017.

[46] Suyog Gupta, Ankur Agrawal, Pritish Narayanan, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision. In *Proceedings of the International Conference on Machine Learning - ICML '15*, pages 1737–1746, 2015.

[47] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv e-print*, 2016.

[48] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv e-print*, 12 2014.

[49] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented Approximation of Convolutional Neural Networks. In *arXiv preprint*, page 8, 2016.

[50] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. In *Advances in Neural Information Processing Systems - NIPS'15*, pages 3123–3131, 2015.

[51] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *Proceedings of the European Conference on Computer Vision - ECCV'16*, pages 525–542, Amsterdam, Netherlands, 2016. Springer.

[52] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, pages 65–74, 2017.

[53] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights. *Proceedings of the IEEE Computer Society Annual Symposium on VLSI - ISVLSI '16*, 2016-Septe:236–241, 2016.

[54] Rui Zhao, Wanli Ouyang, Hongsheng Li, and Xiaogang Wang. Saliency detection by multi-context deep learning. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '15*, pages 1265–1274, 2015.

[55] Hyeonuk Sim and Jongeun Lee. A New Stochastic Computing Multiplier with Application to Deep Convolutional Neural Networks. In *Proceedings of the Annual Conference on Design Automation - DAC '17*, pages 1–6, New York, New York, USA, 2017. ACM.

[56] Vincent T Lee, Armin Alaghi, John P Hayes, Visvesh Sathe, and Luis Ceze. Energy-Efficient Hybrid Stochastic-Binary Neural Networks for Near-Sensor Computing. In *Proceedings of the Conference on Design, Automation and Test in Europe - DATE '17*, 2017.

[57] Sebastian Vogel, Christoph Schorn, Andre Guntoro, and Gerd Ascheid. Efficient Stochastic Inference of Bitwise Deep Neural Networks. *arXiv preprint*, (Nips):1–6, 2016.

[58] Kyounghoon Kim, Jungki Kim, Joonsang Yu, Jungwoo Seo, Jongeun Lee, and Kiyoung Choi. Dynamic Energy-accuracy Trade-off Using Stochastic Computing in Deep Neural Networks. In *Proceedings of the Annual Conference on Design Automation - DAC '16*, number 1, pages 124:1–124:6, New York, NY, USA, 2016. ACM.

[59] Ao Ren, Ji Li, Zhe Li, Caiwen Ding, Xuehai Qian, Qinru Qiu, Bo Yuan, and Yanzhi Wang. SC-DCNN: Highly-Scalable Deep Convolutional Neural Network using Stochastic Computing. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS'17*, pages 405–418, 2017.

[60] Arash Ardakani, Francois Leduc-Primeau, Naoya Onizawa, Takahiro Hanyu, and Warren J. Gross. VLSI Implementation of Deep Neural Network Using Integral Stochastic Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2688–2699, 2015.

[61] Jialiang Zhang and Jing Li. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, pages 25–34, 2017.

[62] R. Tapiador, A. Rios-Navarro, A. Linares-Barranco, Minkyu Kim, Deepak Kadetotad, and Jae-sun Seo. Comprehensive Evaluation of OpenCL-based Convolutional Neural Network Accelerators in Xilinx and Altera FPGAs. *Proceedings of the International Work-Conference on Artificial Neural Networks- IWANN '17*, pages 271–282, 2017.

[63] Jeremy Bottleson, Sungye Kim, Jeff Andrews, Preeti Bindu, Deepak N. Murthy, and Jingyi Jin. ClCaffe: OpenCL accelerated caffe for convolutional neural networks. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium - IPDPS '16*, pages 50–57, 2016.

[64] Intel FPGA. The Intel® FPGA SDK for Open Computing Language (OpenCL), 2016.

[65] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster. In *Proceedings of the International Symposium on Low Power Electronics and Design - ISLPED '16*, pages 326–331, 2016.

[66] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *Proceedings of the International Conference on Computer-Aided Design - ICCAD '16*, pages 1–8, New York, New York, USA, 2016. ACM.

[67] Erwin H. Bareiss. Numerical solution of linear equations with Toeplitz and Vector Toeplitz matrices. *Numerische Mathematik*, 13(5):404–424, 10 1969.

[68] Shmuel Winograd. *Arithmetic complexity of computations*, volume 33. Siam, 1980.

[69] Andrew Lavin and Scott Gray. Fast Algorithms for Convolutional Neural Networks. *arXiv e-print*, arXiv: 150, 9 2015.

[70] Liqiang Lu, Yun Liang, Qingcheng Xiao, and Shengen Yan. Evaluating fast algorithms for convolutional neural networks on FPGAs. In *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines - FCCM '17*, pages 101–108, 2017.

[71] Steven Smith. *The scientist and engineer's guide to digital signal processing*. California Technical Pub. San Diego, 1997.

[72] Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans Peter Graf. A Massively Parallel Coprocessor for Convolutional Neural Networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '17*, pages 53–60. IEEE, 7 2009.

[73] C Farabet, C Poulet, J Y Han, Y LeCun, David R. Tobergte, and Shirley Curtis. CNP: An FPGA-based processor for Convolutional Networks. In *Proceedings of the International Conference on Field Programmable Logic and Applications - FPL '09*, volume 53, pages 1689–1699, 2009.

[74] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A Dynamically Configurable Coprocessor for Convolutional Neural Networks. *ACM SIGARCH Computer Architecture News*, 38(3):247–257, 6 2010.

[75] C Farabet, B Martini, B Corda, P Akselrod, E Culurciello, and Y LeCun. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '11*, pages 109–116, 6 2011.

[76] Vinayak Gokhale, Jonghoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. A 240 G-ops/s mobile coprocessor for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '14*, pages 696–701, 6 2014.

[77] Rahman Atul, Lee Jongeun, and Choi Kiyoung. Efficient FPGA acceleration of Convolutional Neural Networks using logical-3D compute array. In *Proceedings of the Conference on Design, Automation and Test in Europe - DATE '16*, Dresden, Germany, 2016. IEEE.

[78] Yufei Ma, Naveen Suda, Yu Cao, Jae Sun Seo, and Sarma Vrudhula. Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA, 2016.

[79] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer CNN accelerators. In *Proceedings of the Annual International Symposium on Microarchitecture - MICRO '16*, volume 2016-Decem, 2016.

[80] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, pages 45–54, 2017.

[81] Steven Derrien and Sanjay Rajopadhye. Loop tiling for reconfigurable accelerators. In *Proceedings of the International Conference on Field Programmable Logic and Applications - FPL '01*, volume 2147, pages 398–408. Springer, 2001.

[82] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65, 4 2009.

[83] Yufei Ma, Minkyu Kim, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. End-to-end scalable FPGA accelerator for deep residual networks. In *Proceedings of the IEEE International Symposium on Circuits and Systems - ISCAS '17*, pages 1–4. IEEE, 5 2017.

[84] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *Proceedings of the International Conference on Field Programmable Logic and Applications - FPL '17*, pages 1–8. IEEE, 9 2017.

[85] Zhiqiang Liu, Yong Dou, Jingfei Jiang, Jinwei Xu, Shijie Li, Yongmei Zhou, and Yingnan Xu. Throughput-Optimized FPGA Accelerator for Deep Convolutional Neural Networks. *ACM Transactions on Reconfigurable Technology and Systems*, 10(3):1–23, 2017.

[86] Jack B Dennis and David P Misunas. A Preliminary Architecture for a Basic Data-flow Processor. In *Proceedings of the International Symposium on Computer Architecture - ISCA '75*, pages 126–132. ACM, 1975.

[87] Li Lin, Tiziana Fanni, Timo Viitanen, Xie Renjie, Francesca Palumbo, Luigi Raffo, Heikki Huttunen, Jarmo Takala, and Shuvra S Bhattacharyya. Low power design methodology for signal processing systems using lightweight dataflow techniques. In *Proceedings of the Conference on Design and Architectures for Signal and Image Processing - DASIP' 16*, pages 82–89. IEEE, 10 2016.

[88] Chung-Ching Shen, William Plishker, Hsiang-Huang Wu, and Shuvra S Bhattacharyya. A lightweight dataflow approach for design and implementation of SDR systems. In *Proceedings of the Wireless Innovation Conference and Product Exposition*, pages 640–645, 2010.

[89] Edward A Lee and David G Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, 1987.

[90] Stylianos I Venieris and Christos Savvas Bouganis. Latency-Driven Design for FPGA-based Convolutional Neural Networks. In *Proceedings of the International Conference on Field Programmable Logic and Applications - FPL '17*, 2017.

[91] Fengfu Li, Bo Zhang, and Bin Liu. Ternary Weight Networks. *arXiv e-print*, 5 2016.

[92] Sparsh Mittal. A Survey of Techniques for Approximate Computing. *ACM Computing Surveys*, 48(4):1–33, 3 2016.

[93] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Fixed point optimization of deep convolutional neural networks for object recognition. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '15*, 2015.

[94] Darryl Lin, Sachin Talathi, and V Annapureddy. Fixed Point Quantization of Deep Convolutional Networks. In *Proceedings of the International Conference on Machine Learning - ICML '16*, pages 2849 – 2858, 2016.

[95] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *arxiv e-print*, 9 2016.

[96] Shuchang Zhou, Yuzhi Wang, He Wen, Qinyao He, and Yuheng Zou. Balanced Quantization: An Effective and Efficient Approach to Quantized Neural Networks. *Journal of Computer Science and Technology*, 32:667–682, 2017.

[97] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized Convolutional Neural Networks for Mobile Devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '16*, pages 4820–4828, 2016.

[98] Jean-Pierre David, Kassem Kalach, and Nicolas Tittley. Hardware Complexity of Modular Multiplication and Exponentiation. *IEEE Transactions on Computers*, 56(10):1308–1319, 10 2007.

[99] D. Williamson. Dynamically scaled fixed point arithmetic. In *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing Conference*, pages 315–318. IEEE, 1991.

[100] Shasha Guo, Lei Wang, Baozi Chen, Qiang Dou, Yuxing Tang, and Zhisheng Li. FixCaffe: Training CNN with Low Precision Arithmetic Operations by Fixed Point Caffe. In *Proceedings of the International Workshop on Advanced Parallel Processing Technologies - APPT '17*, pages 38–50. Springe, 8 2017.

[101] Hiroki Nakahara, Tomoya Fujii, and Shimpei Sato. A fully connected layer elimination for a binarizec convolutional neural network on an FPGA. In *Proceedings of the International Conference on Field Programmable Logic and Applications - FPL '17*, pages 1–4. IEEE, 9 2017.

[102] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems - NIPS'16*, pages 4107–4115, 2 2016.

[103] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained Ternary Quantization. In *Proceedings of the International Conference on Learning Representations - ICLR'17*, 12 2017.

[104] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, 2017.

[105] Nicholas J Fraser, Yaman Umuroglu, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Scaling Binarized Neural Networks on Reconfigurable Logic. In *Proceedings of the Workshopon Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms - PARMA-DITAM '17*, pages 25–30. ACM, 2017.

[106] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. FP-BNN: Binarized Neural Network on FPGA. *Neurocomputing*, 10 2017.

[107] Adrien ProstBoucle, Alban Bourge, Frédéric Pétrot, Hande Alemdar, Nicholas Caldwell, and Vincent Leroy. Scalable High-Performance Architecture for Convolutional Ternary Neural Networks on FPGA. In *Proceedings of the International Conference on Field Programmable Logic and Applications - FPL '17*, 7 2017.

[108] Armin Alaghi and John P Hayes. Fast and Accurate Computation using Stochastic Circuits. In *Proceedings of the Conference on Design, Automation and Test in Europe - DATE '14*. IEEE, 2014.

[109] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '15*, pages 806–814, 2015.

[110] Song Han, Jeff Pool, John Tran, and William J Dally. Learning both Weights and Connections for Efficient Neural Network. In *Advances in Neural Information Processing Systems - NIPS'15*, pages 1135–1143, 2015.

[111] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '17*, 2017.

[112] Song Han, Huizi Mao, and William J. Dally. Deep Compression - Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *Proceedings of the International Conference on Learning Representations - ICLR'16*, pages 1–13, 2016.

[113] Amos Sironi, Bugra Tekin, Roberto Rigamonti, Vincent Lepetit, and Pascal Fua. Learning separable filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(1):94–106, 2015.

[114] Richard Dorrance, Fengbo Ren, and Dejan Marković. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '14*, pages 161–170, New York, New York, USA, 2014. ACM.