

Using Data Compression for Optimizing FPGA-Based Convolutional Neural Network Accelerators

Yijin Guan¹(✉), Ningyi Xu², Chen Zhang¹, Zhihang Yuan¹,
and Jason Cong^{1,3}

¹ Center for Energy-Efficient Computing and Applications, PKU, Beijing, China
guanyijin@pku.edu.cn

² Microsoft Research Asia, Beijing, China

³ Computer Science Department, University of California, Los Angeles, USA

Abstract. Convolutional Neural Network (CNN) has been extensively employed in research fields including multimedia recognition, computer vision, etc. Various FPGA-based accelerators for deep CNN have been proposed to achieve high energy-efficiency. For some FPGA-based CNN accelerators in embedded systems, such as UAVs, IoT, and wearable devices, their overall performance is greatly bounded by the limited data bandwidth to the on-board DRAM. In this paper, we argue that it is feasible to overcome the bandwidth bottleneck using data compression techniques. We propose an *effective roofline model* to explore design trade-off between computation logic and data bandwidth after applying data compression techniques to parameters of CNNs. We implement a decompression module and a CNN accelerator on a single Xilinx VC707 FPGA board with two different compression/decompression algorithms as case studies. Under a scenario with limited data bandwidth, the peak performance of our implementation can outperform designs using previous methods by 3.2× in overall performance.

Keywords: CNN · FPGA · Compression/decompression

1 Introduction

Convolutional Neural Network (CNN) [9], a popular deep learning algorithm, has become the most successful algorithm for visual content understanding, image search, and classification [6, 8]. In recent years, CNN has achieved great improvement on both neural network architecture and accuracy, which makes CNN outperform conventional approaches. However, previous research has demonstrated that general purposed processors like CPUs are not efficient to perform the computation of CNN algorithms. As a result, various accelerators for CNN have been proposed recently. Among these accelerators, FPGA-based CNN accelerators have attracted great attention because of their high performance, low power consumption (compared with CPUs), and flexibility [1, 5, 10, 11, 16].

Previous works on FPGA-based CNN accelerator aim at optimizing computation throughput [1, 5, 11] and I/O bandwidth [10] to achieve the best performance. In [16], Zhang et al. proposed a roofline model to find the design solution with the highest performance and lowest bandwidth requirements. The model can help find an optimal design configuration under the constraints of computation roof and bandwidth roof, which are provided by the specific hardware platform. More details can be found in Sect. 2.2.

Having this model, it is also easy to tell whether computation resource or I/O bandwidth has become the bottleneck of an FPGA-based CNN accelerator. In fact, in most of modern embedded systems, such as UAVs, mobile phones, IoT and wearable devices, the I/O bandwidth limitation (commonly 100–200 MB/s) is even stricter, which further lowers the bandwidth roof and results in a decrease on the overall performance of the CNN accelerators.

To overcome the problem of limited bandwidth, we further explore trade-off between computation resource and data bandwidth with consideration of compression techniques. In particular, we notice that the number of parameters (weights and bias) in real-life CNN is usually too large to be stored on-chip (e.g. about 60 million and 140 million of parameters for AlexNet [7] and VGG [12] respectively), which indicates that users need to load parameters from external storage to computation engines for CNN computation. Besides, the parameters of CNN are pre-calculated off-line in training phase, and they remain the same during inference phase. Taking advantage of this characteristic, we can compress these parameters off-line in advance, and only decompress them on-line on FPGA for CNN computation. While applied in embedded systems, CNN only performs the inference phase in various real-life applications, so we focus on a real-time acceleration for the inference phase of CNN.

To find the optimal design, we propose an effective roofline model. Consequently, we can further improve performance and even reduce energy consumption under the same bandwidth constraint. Moreover, we also provide analysis on the design space exploration and characteristics of different compression/decompression algorithms. To the best of our knowledge, this is the first work on applying compression/decompression methods to the parameters of CNN to improve the bandwidth bottleneck.

The main contributions of this work are summarized as follows,

- We build an effective roofline model for problem formulation and performance analysis, which takes both CNN accelerator and decompression module into consideration.
- We present a method to find the optimal configuration for architecture design, with a best on-chip resource allocation between CNN accelerator and decompression module using the effective roofline model.
- As case studies, we implement decompression modules using two typical compression/decompression algorithms, which improve the performance of CNN accelerator by $2.37\times$ and $3.20\times$ respectively, while saving energy at the same time.

The rest of this paper is organized as follows: Sect. 2 introduces CNN and roofline model, and Sect. 3 explains our methodology for performance optimization. Section 4 presents our hardware implementation. Experimental results and analysis are shown in Sect. 5. Section 6 concludes this paper and discusses about future work.

2 Background

In this section, we first introduce some basic concepts of CNN and explain our ideas generally. Then we present the roofline model for performance analysis in previous work.

2.1 CNN Basis

CNN is a classical supervised learning algorithm, and has achieved state-of-the-art accuracy across a broad set of applications. Typically, CNN is composed of two kinds of layers: convolutional layers (feature extractor) and fully connected layers (classifier).

A typical convolutional layer is shown in Fig. 1. As this figure illustrates, several feature maps form the input of a convolutional layer. These input feature maps are filtered by their own convolution kernels, then we can get a set of filtered feature maps as the output. Each convolution kernel is composed of many parameters, also called weights and bias. Deploying a CNN normally includes two phases: training and inference. In practice, training is accomplished off-line using a cluster of CPUs [4] or GPUs [2, 14, 15], and parameters are adjusted in a backward direction to get the best accuracy with a training set. During inference phase, the trained CNN is deployed for real-life applications, and computation executes in a forward direction on-line. So the speed of inference is the key factor of CNN’s overall performance, and we focus on accelerating the inference phase in this work. It is worth noting that parameters remain unchanged during inference, which provides us with the possibility of compressing them off-line before they are applied to real-life applications, and only doing the decompression work on-line.

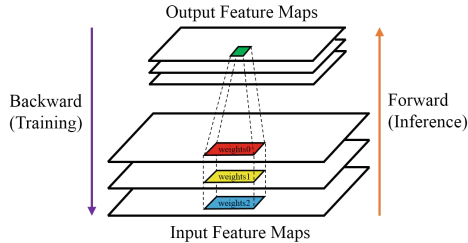


Fig. 1. Overview of a convolutional layer

Work in [3] has demonstrated that convolutional operations will occupy over 90% of the computation time of a CNN during the inference phase, so we focus on accelerating convolutional layers in this work, and discuss about fully connected layers in Sect. 6.

2.2 Roofline Model

Roofline Model is first introduced in [13] to restrict system performance under the highest attainable performance and data accessing bandwidth provided by a specific platform. Figure 2 shows an example of roofline model.

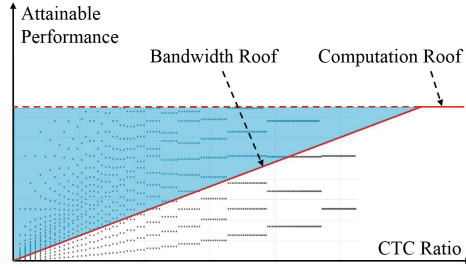


Fig. 2. Performance analysis using roofline model

As shown in Fig. 2, in roofline model, X-axis is computation to communication ratio (*CTC Ratio*), which indicates the number of computation operations per I/O traffic. Y-axis is the attainable performance (*AP*) of a design in GOPS (Giga operations per second). Here we denote the number of computation operations in CNN accelerator as *Operations*, and denote the amount of external data access for computation as *Data*. So we can calculate *CTC Ratio* and *AP* according to Eq. 1. According the definitions of *CTC Ratio* and *AP*, we can calculate the required bandwidth (BW_r) of a possible design by Eq. 2.

Roofline model defines computation roof to represent the peak performance that utilizes all the computation resources, and it also defines bandwidth roof, whose slope equals to the maximum data accessing bandwidth provided by the hardware platform (denoted by BW). On this hardware platform, the highest performance that the accelerator can achieve is restricted by computation roof and bandwidth roof. This can be summarized in Eq. 3.

$$CTC\ Ratio = \frac{Operations}{Data}, \quad AP = \frac{Operations}{Cycles} \quad (1)$$

$$BW_r = \frac{Data}{Cycles} = \frac{AP}{CTC\ Ratio} \quad (2)$$

$$AP_{max} = \min(Computation\ Roof, CTC\ Ratio * BW) \quad (3)$$

3 Methodology

3.1 Effective Roofline Model

Inspired by roofline model, we propose an effective roofline model for performance optimization. Applying decompression module to CNN accelerator brings some changes to the formulations in Sect. 2.2. We denote the compression ratio as r (Eq. 4). For a single decompression unit, we denote its throughput as BW_d , which equals to the amount of data that the decompression unit can output in one second. However, a single decompression unit may not satisfy our demand for maximized resource utilization and higher performance, so we duplicate decompression unit according to the resources on chip, which offers great conciseness and flexibility to our adjustment of resource utilization and speed of decompression. Here we denote the number of duplications as n . In fact, the data size of input enough ignored when compared with the huge amount of parameters to be loaded during inference phase. So the Attainable Performance and CTC Ratio after applying a decompression module can be calculated by Eqs. 5 and 6 respectively.

$$r = \frac{\text{Size of Compressed Data}}{\text{Size of Original Data}} \quad (4)$$

$$CTC \text{ Ratio}' = \frac{\text{Operations}}{\text{Data}'} = \frac{\text{Operations}}{\text{Data} * r} \quad (5)$$

$$AP' = \frac{\text{Operations}}{\text{Cycles} + \text{Cycles of Decompression}} = \frac{\text{Operations}}{\text{Cycles} + \frac{\text{Data}}{n * BW_d}} \quad (6)$$

To find the best design configuration under roofline model, we need to calculate the new locations of all the design points again every time the value of n changes. As a result, the amount of overall computation for estimation is highly increased, which makes it more difficult to find the best design configuration. So we propose to solve this problem in another easier and clearer way.

According to Sect. 2.2, we denote the I/O bandwidth provided by the platform as BW . While applying a decompression module between storage and CNN accelerator, the I/O bandwidth that the CNN accelerator actually obtains varies, we denote it as BW' . Based on the definitions above, the relationship between BW' and BW is shown in Eqs. 7 and 8. BW is determined by the specific platform. r and BW_d are determined by the compression/decompression algorithms and hardware implementations respectively. n is the variant to reflect the trade-off between resources for decompression module and resources for CNN accelerator.

$$\text{When } n = 0, BW' = BW \quad (7)$$

$$\text{When } n > 0, BW' = \frac{BW}{r + \frac{BW}{n * BW_d}} \quad (8)$$

With the formulations above, we present an effective roofline model to solve the highly complex problem that the decompression module brings.

Figure 3 shows an example of our effective roofline model. In effective roofline model, we define an effective computation roof (ECR) as the highest attainable performance of CNN accelerator with the on-chip resources that can be used for it, and we also define an effective bandwidth roof (EBR), whose slope equals to BW' .

Deploying a decompression module has two aspects of influence on the CNN accelerator: On the one hand, the decompression module definitely occupies a certain amount of resources, which may decrease the on-chip resources available for CNN accelerator. As n increases, the resources for CNN accelerator may further decrease, which results in a decrease on the attainable performance. This can be reflected as a downwards movement of ECR . On the other hand, according to Eq. 8, BW' will increase when n increases, which results in a anticlockwise movement of EBR in effective roofline model. Therefore, for different choices of values for n ($n_0 < n_1 < n_2$), the corresponding $ECRs$ and $EBRs$ are shown in Fig. 3.

As a result, the design space of the roofline model introduced in [16] is just a subset (when $n = 0$) of the design space of effective roofline model. After adding the decompression module, our effective roofline model takes computation power, bandwidth requirements and on-chip resource allocation into consideration, so it can explore a much larger design space and probably find a design configuration with better overall performance.

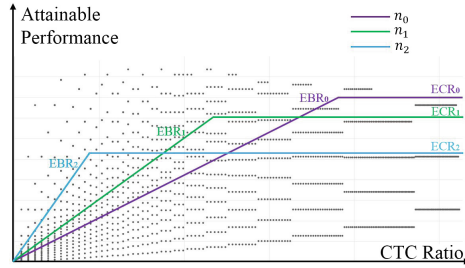


Fig. 3. An example of effective roofline model

3.2 Design Space Exploration

Taking advantage of the characteristics and parameterization of CNN accelerator, every possible design can be represented as a point in the effective roofline model. All these points comprise a huge space of possible design choices, and we propose a method to efficiently find the design with highest overall performance.

According to our effective roofline model, when n equals to an arbitrary value, the method to find the best design configuration is similar to that in conventional roofline model. Under the constraints of ECR and EBR , we can use a traversal approach to find the optimal configuration for architecture design with highest performance and lowest bandwidth requirements, and this method

has been presented in [16]. Every time n changes, ECR and EBR will change, which means that we need to search for the best design among all the points in all possible values of n . To simplify this procedure, we use pruning methods to shrink the searching space.

On the one hand, when $n = 0$, which means we do not apply decompression module to the CNN accelerator, we have $BW' = BW$. Using the method provided in [16], we can find a point (X in Fig. 4) with the best performance. Then we add decompression module to this system to search for a point with better overall performance. So if there exists such a point that is better than X, this point must be located at the left side of $EBR_{n=0}$ and at the upside of X's attainable performance. On the other hand, when we increase n to further improve bandwidth bottleneck, ECR may move downwards. Supposing ECR equals to X's attainable performance when n equals to a certain value (denoted by n_{max}), then there is no need to further increase n . Above all, we need to traverse n from 1 to n_{max} to search for the best trade-off in resource allocation. For each value of n , we only need to search for the best design among the points in the shaded region (shown in Fig. 4) instead of the entire design space.

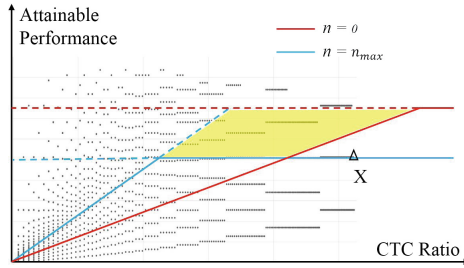


Fig. 4. An example of design space exploration

4 Implementation

4.1 System Design

The system design is shown in Fig. 5. We divide the whole function of this system into two parts: Compression and CNN-D (CNN accelerator with decompression module). The arrows in Fig. 5 show the direction of parameter flow. White arrows indicate that the parameters transferred are compressed, while black arrows indicate that the parameters transferred are decompressed.

As Fig. 5 illustrates, Compression is mainly implemented on software. The Compression Module is used to compress the parameters of our implemented CNN, and Dispatcher is deployed to dispatch them into the format suitable for parallel decompression. To emulate the bandwidth bounded scenario in embedded systems, we attach a NAND Flash chip to our FPGA board, and this NAND Flash chip works as the external storage where the parameters of CNN are stored.

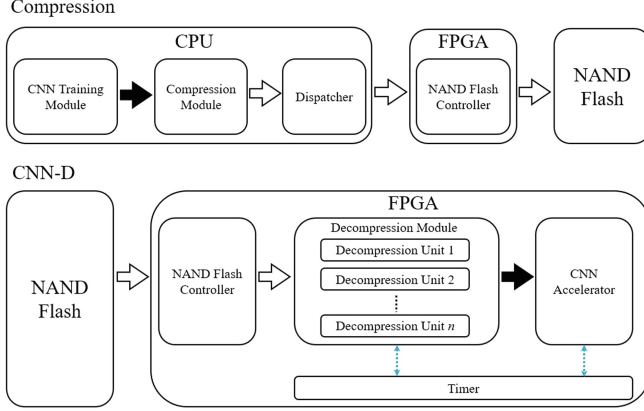


Fig. 5. Overview of system design

Our whole design of CNN-D is implemented on a single FPGA board. The NAND Flash Controller works as a data fetcher and data dispatcher for Decompression Module. It fetches parameters stored in the NAND Flash and dispatches parameters to each Decompression Unit. The Decompression Module is composed of n Decompression Units, and each Decompression Unit decompresses the parameters transferred into it. After decompression, the decompressed parameters are transferred to the CNN Accelerator, where the main part of CNN computation is performed. What is more, we use a Timer to measure the execution time of our design.

4.2 Compression/Decompression Algorithms

Applying compression/decompression modules to minimize the amount of data to be transferred is a common approach in system design for bandwidth optimization. However, there is something different for our demand on the compression/decompression algorithms. Firstly, we do not care how much time and resources it costs to compress parameters of CNN, since we compress them offline only once, and store them in a read-only mode. Secondly, we hope decompression does not cost much time and resources considering the performance of the whole CNN accelerator. In summary, our requirements to the compression/decompression algorithms are: high compression ratio, high decompression speed and low resource utilization for decompression. Considering representativeness and our requirements, we choose LZ77 as an example of dictionary based algorithms, and Huffman Encoding as an example of entropy encoding based algorithms. Many compression/decompression algorithms used nowadays are variants or combinations of these two algorithms.

4.3 CNN Accelerator

The implementation of CNN accelerator is generally shown in Fig. 6. All the computation of CNN are accomplished in parallel by numerous convolution units. Several optimizations are applied to the design of convolution units, such as deep pipelining, loop unrolling and loop tiling. For data access optimization, we implement two data buffers for data reusing and ping-pong operations. All these optimization strategies can be parameterized, which makes it possible to calculate the *CTC Ratio* and *AP* of each design configuration accurately. For the choice of optimization parameters, we refer to the best design configuration found by our effective roofline model.

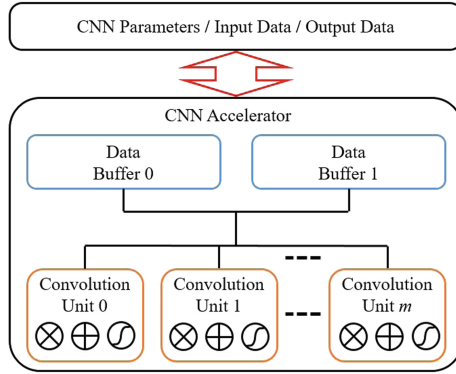


Fig. 6. Overview of CNN accelerator

5 Case Study

In this section, the experimental setup of our experiments is provided first. Then we present and analyze the experimental results.

5.1 Experimental Setup

We use Vivado HLS (v2015.4) to implement our CNN accelerator and decompression module. Vivado HLS is a high level synthesis design tool, which takes C code as input and outputs IP core in Verilog HDL. For the design space exploration and performance estimation, we use the pre-synthesis report of Vivado HLS. Then the RTL synthesis and implementation are done in Vivado (v2015.4).

The hardware platform we choose is a VC707 board with a Xilinx Virtex7 485t FPGA chip on it, and its working frequency is set to be 100 MHz. The storage device we use is SAMSUNG K9F1G08U0D NAND Flash board.

To test our effective roofline model in a real-life case, we implement a CNN with our accelerators, VGG-19 [12], which has 16 convolutional layers. The VGG

Model increases depth using an architecture with very small (3×3) convolution kernels, which shows that a significant improvement on the prior-art configurations can be achieved by pushing the depth to 16–19 layers. The detailed configurations of VGG-19 can be found in [12]. The input is a 224×224 RGB image, and the convolution kernel size in convolutional layers is 3×3 , with a sliding stride of 1.

5.2 Experimental Results

Table 1 shows the average compression ratio (r in Sect. 3.1, $r < 1$), speed of decompression (BW_d in Sect. 3.1) and resource utilization of different decompression units. From Table 1 we can see that Huffman Encoding performs about $1.30\times$ better than LZ77 on average compression ratio. This is because that LZ77 is a dictionary-based compression algorithm, and performs better when the data has a stronger locality. However, parameters of VGG-19 show a weak locality. In other CNN models, the locality of parameters varies, LZ77 may perform better. This is also the reason why we implement two different typical compression algorithms. According to analysis in Sect. 3.1, decompression speed of a single decompression unit is not very important for our application, since we can adjust n for different BW' , and CNN computation is the dominating factor. As shown in Table 1, a single decompression unit does not occupy much resource. More specifically, the main kind of resource these decompression units occupy is LUT, and they do not use DSP at all. While computation resource (DSP) is crucial to the performance of CNN accelerator, so this means a greater space for our optimization.

Table 1. Decompression unit comparison

Algorithm	r	BW_d	DSP	BRAM	LUT	FF
LZ77	0.48	114.7 MB/s	0.00%	0.97%	4.52%	0.82%
Huffman	0.37	90.61 MB/s	0.00%	0.49%	1.04%	0.16%

We implement three cases for our studies: design with no decompression module (named as *CNN*), design combined with LZ77 decompression module (named as *CNN-D(LZ)*) and design combined with Huffman Encoding decompression module (named as *CNN-D(HE)*). All implementations implement the best hardware configuration found by the method presented in Sect. 3.2. The bandwidth of data accessing is 181.20 MB/s, which is within the typical bandwidth range (100–200 MB/s) in real-life embedded systems.

The overall resource utilization of these three designs are shown in Table 2. As shown in Table 1, decompression units occupy much more LUT and FF than DSP and BRAM. When we duplicate decompression units to achieve better performance, the demand for LUT and FF increases greatly. As a result, compared with *CNN* in Table 2, we can observe a significant increase on the utilization of LUT and FF in *CNN-D(LZ)* and *CNN-D(HE)*.

Table 2. Overall resource utilization

Implementation	DSP	BRAM	LUT	FF
<i>CNN</i>	10.00%	6.25%	8.66%	5.23%
<i>CNN</i> – <i>D(LZ)</i>	27.14%	17.48%	85.36%	32.78%
<i>CNN</i> – <i>D(HE)</i>	40.00%	30.10%	89.92%	18.39%

Table 3. Performance comparison

Number of layer	<i>CNN</i>		<i>CNN</i> – <i>D(LZ)</i>		<i>CNN</i> – <i>D(HE)</i>	
	Time (s)	GOPS	Time (s)	GOPS	Time (s)	GOPS
1	0.061	5.69	0.031	11.19	0.031	11.19
2	1.31	5.65	0.66	11.21	0.65	11.38
3	0.49	7.55	0.16	23.12	0.16	23.12
4	0.98	7.55	0.33	22.42	0.33	22.42
5	0.41	9.02	0.16	23.12	0.082	45.11
6, 7, 8	0.82	9.02	0.33	22.42	0.16	46.24
9	0.37	10.00	0.16	23.12	0.12	30.83
10, 11, 12	0.73	10.14	0.33	22.42	0.24	30.83
13, 14, 15, 16	0.18	10.07	0.082	22.55	0.061	30.21
Overall GOPS	8.66		20.49		27.69	
Speedup	1.00×		2.37×		3.20×	

The performance comparison is shown in Table 3. Since the configurations of some convolutional layers in VGG-19 are the same, their results are shown in a single row. We show the results of convolutional layers only, because convolutional operations occupy most of the computation time of a CNN during the inference phase, which has been discussed about in Sect. 2.1.

The overall performance of *CNN* is only 8.66 GOPS, which is pretty bad if compared with previous designs. For example, design in [16] can achieve an higher overall performance of 61.62 GOPS. However, it is worth noticing that the bandwidth roof of data accessing in *CNN* is limited to 181.20 MB/s, which is within the typical bandwidth range (100–200 MB/s) in real-life embedded systems, while in design of [16], the bandwidth roof is 4.5 GB/s. So the obvious difference of overall performance proves our claim that limited bandwidth in embedded systems becomes a strict bound that prevents CNN accelerator from achieving a higher performance.

Compared with *CNN*, we can see that *CNN* – *D(LZ)* achieves 2.37× speedup in overall performance, and the speedup that *CNN* – *D(HE)* achieves is 3.20×. Since the change of runtime power of our FPGA board due to changes of resource utilization is slight enough to be ignored, we can save almost the same ratio of energy as that of speedups.

6 Conclusions and Future Work

In this paper, we propose to use data compression to further improve the overall performance of FPGA-based CNN accelerators. We present an effective roofline model to solve the resource trade-off between decompression module and CNN accelerator. This effective roofline model formulates a more general scenario and includes the design space of former CNN accelerator works. In addition, we shrink the design space for exploration, and provides a method to find the optimal design configuration. Finally, we implement the system on a Xilinx VC707 FPGA board, which achieved great improvement upon implementations using previous methods.

We are working on extension of this work in several directions. First of all, we use LZ77 and Huffman Encoding in our case studies. Lossy compression/decompression algorithms are not taken into consideration. We expect that, in the near future, we can come up with an accurate model to describe the key characteristics of different compression/decompression algorithms. What is more, this model can be combined with our effective roofline model for a better modeling and estimation. Secondly, Artificial Neural Network (ANN) is composed of fully connected layers only, which indicates more parameters to be transferred. Though the computation pattern of ANN is a little different from that of CNN, our proposed effective roofline model can still work with a few modifications. We plan to analyze several real-life ANNs applied in embedded systems, and test how much improvement we can achieve with the help of effective roofline model.

References

1. Cadambi, S., Majumdar, A., Becchi, M., Chakradhar, S., Graf, H.P.: A programmable parallel accelerator for learning and classification. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, pp. 273–284. ACM (2010)
2. Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., Andrew, N.: Deep learning with COTS HPC systems. In: Proceedings of the 30th International Conference on Machine Learning, pp. 1337–1345 (2013)
3. Cong, J., Xiao, B.: Minimizing computation in convolutional neural networks. In: Wermter, S., Weber, C., Duch, W., Honkela, T., Koprinkova-Hristova, P., Magg, S., Palm, G., Villa, A.E.P. (eds.) ICANN 2014. LNCS, vol. 8681, pp. 281–290. Springer, Cham (2014). doi:[10.1007/978-3-319-11179-7_36](https://doi.org/10.1007/978-3-319-11179-7_36)
4. Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q.V., et al.: Large scale distributed deep networks. In: Advances in Neural Information Processing Systems, pp. 1223–1231 (2012)
5. Farabet, C., Poulet, C., Han, J.Y., LeCun, Y.: CNP: an FPGA-based processor for convolutional networks. In: International Conference on Field Programmable Logic and Applications, FPL 2009, pp. 32–37. IEEE (2009)
6. Ji, S., Xu, W., Yang, M., Yu, K.: 3D convolutional neural networks for human action recognition. IEEE Trans. Pattern Anal. Mach. Intell. **35**(1), 221–231 (2013)

7. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: *Advances in Neural Information Processing Systems*, pp. 1097–1105 (2012)
8. Larochelle, H., Erhan, D., Courville, A., Bergstra, J., Bengio, Y.: An empirical evaluation of deep architectures on problems with many factors of variation. In: *Proceedings of the 24th International Conference on Machine Learning*, pp. 473–480. ACM (2007)
9. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. *Proc. IEEE* **86**(11), 2278–2324 (1998)
10. Peemen, M., Setio, A., Mesman, B., Corporaal, H., et al.: Memory-centric accelerator design for convolutional neural networks. In: *IEEE 31st International Conference on Computer Design (ICCD)*, pp. 13–19. IEEE (2013)
11. Sankaradas, M., Jakkula, V., Cadambi, S., Chakradhar, S., Durdanovic, I., Cosatto, E., Graf, H.P.: A massively parallel coprocessor for convolutional neural networks. In: *20th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2009*, pp. 53–60. IEEE (2009)
12. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. *arXiv preprint [arXiv:1409.1556](https://arxiv.org/abs/1409.1556)* (2014)
13. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* **52**(4), 65–76 (2009)
14. Yadan, O., Adams, K., Taigman, Y., Ranzato, M.: Multi-GPU training of convnets. *arXiv preprint [arXiv:1312.5853](https://arxiv.org/abs/1312.5853)*, p. 17 (2013)
15. Yu, K.: Large-scale deep learning at Baidu. In: *Proceedings of the 22nd ACM International Conference on Conference on Information and Knowledge Management*, pp. 2211–2212. ACM (2013)
16. Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., Cong, J.: Optimizing FPGA-based accelerator design for deep convolutional neural networks. In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170. ACM (2015)

Advanced Parallel Processing Technologies

12th International Symposium, APPT 2017, Santiago de Compostela, Spain, August 29, 2017, Proceedings

Dou, Y.; Lin, H.; Sun, G.; Wu, J.; Heras, D.; Bougé, L. (Eds.)

2017, IX, 129 p. 74 illus., Softcover

ISBN: 978-3-319-67951-8