2017 IEEE Computer Society Annual Symposium on VLSI

On how to design dataflow FPGA-based accelerators for Convolutional Neural Networks

Giuseppe Natale, Marco Bacis, Marco Domenico Santambrogio
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milano, Italy
{giuseppe.natale, marco.santambrogio}@polimi.it
marco.bacis@mail.polimi.it

***Abstract*—In the past few years we have experienced an extremely rapid growth of modern applications based on deep learning algorithms such as Convolutional Neural Network (CNN), and consequently, an intensification of academic and industrial research focused on the optimization of their implementation. Among the different alternatives that have been explored, FPGAs seems to be one of the most attractive, as they are able to deliver high performance and energy-efficiency, thanks to their inherent parallelism and direct hardware execution, while retaining extreme flexibility due to their reconfigurability.**

In this paper we present a design methodology of a dataflow accelerator for the implementation of CNNs on FPGAs, that ensures scalability – and achieve a higher degree of parallelism as the size of the CNN increases – and an efficient exploitation of the available resources. Furthermore, we analyze resource consumption of the layers of the CNN as well as latency in relation to the implementation's hyperparameters. Finally, we show that the proposed design implements a high-level pipeline between the different network layers, and as a result, we can improve the latency to process an image by feeding the CNN with batches of multiple images.

***Index Terms*—Field Programmable Gate Arrays, Convolutional Neural Networks, Dataflow Architectures**

I. Introduction

In light of the successes demonstrated by deep learning techniques, we are recently assisting to their massive development and diffusion, up to a point where we take advantage of them in our every-day life. Among the different deep learning algorithms, Convolutional Neural Network (CNN) have demonstrated to be successful in many domains, such as video surveillance, mobile robot vision, and as image search engines in data centers[9, 10, 12, 11], achieving far higher accuracy than traditional algorithms for computer vision. They have indeed become the state-of-the-art for visual recognition and classification, attracting interests from both industry and academia [18, 6]. CNN mimics the receptive field of biological neurons in the Primary Visual Cortex, by applying consecutive convolution filters to extract features at increasing levels of abstraction, used then for classification.

Along with the improvements in accuracy and the refinements of their models, CNN implementations are also increasing their network size and computation requirements, posing a hard challenge on modern CPUs, not really able to met the needed performance and energy-efficiency requirements. Research has therefore focused on exploring specialized hardware accelerators based on GPUs, ASICs and FPGAs, achieving superior performance figures, with a better energy-efficiency[5, 20, 16, 15]. CNN acceleration using FPGAs is particularly interesting, as FPGAs offers high-flexibility due to their programmability, and high performance and low power consumption thanks to direct hardware execution, effectively offering an appealing trade-off between the best features of GPUs and ASICs[2, 7, 8, 4].

The literature of CNN acceleration using FPGA mostly focuses on the acceleration of the sole convolution layers, due to their heavy impact on the overall computation[20, 8, 4]. In such situation, the chosen approach induce a substantial communication overhead caused by the continuous need for data exchange between the accelerator, that is employed to implement only part of the network, and the host processor, reducing the effectiveness of off-loading the computation to a specialized processor. Others, like [19], attempts at creating a framework for the acceleration of the whole CNN. However, they propose acceleration methodologies that result in a suboptimal exploitation of the on-chip memory, and an under-utilization of the inherent parallelism of the FPGA. Also, they are not able to exploit the likelihood – as it often happens in real-world applications – of CNNs being employed to process batches of multiple images, taking advance of such situation to extract more efficiency. We instead propose in this paper an acceleration methodology of CNN via FPGA, consisting of a dataflow architecture that is able to scale up the size of each layer from single-input-port/single-output-port to fully parallel (on the condition that enough resources are available) and is able to implement a high-level pipeline between the different network layers, as will be shown in the evaluation Section. Such methodology derives from previous work on the acceleration of Iterative Stencil Loops (ISLs)[3, 13], a class of algorithms whose computational pattern share similarities with convolutions. To the best of our knowledge, there are no FPGA-based acceleration methodologies that fully exploit the dataflow computation pattern of CNNs, provide a scalable implementation of each layer, and are able to implement a high-level pipeline within the layers that can improve the throughput when processing batches of multiple images.

The rest of the paper is organized as follows. Section II

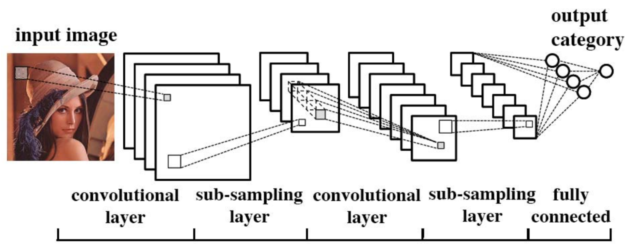

Fig. 1. Feed-forward of a Convolutional Neural Network.

presents the necessary background. Section III describes the proposed acceleration methodology and discusses the implementation choices of a complete network. Then, Section IV presents the experimental evaluation. We end the paper with the concluding remarks and discuss the future work in Section V.

II. Background

A. Convolutional Neural Network

Convolutional Neural Network (CNN) is a machine learning algorithm extended from Artificial Neural Network (ANN) and specifically tailored for image analysis and other similar 2D-structured data. Its structure, depicted in Figure 1, consists of a chain of multiple layers that first extract more and more complex features, collected in the *feature maps*, from the image (*feature extraction stage*), and then classify it (*classification stage*).

The **features extraction** stage is defined as a chain of two different layers: the convolutional layer, and the sub-sampling (or pooling) layer.

The **convolutional layer** applies a series of K filters (or kernels) to the previous features maps. In particular, for each filter $k \in K$, the convolutional layer computes the following equation:

$$o_{i,j,k} = \sum_{h=0}^{H_k} \sum_{m=0}^{W_k} \sum_{c=0}^{C_k} (w_{h,m,c} \cdot x_{i+h,j+m,c}) + b_k \qquad (1)$$

where H_k and W_k represents respectively height and weight of the kernel, C refers to the number of feature maps produced by the previous layer ($C_k \leq C$), b_k is a bias, i and j are the coordinates of the current pixel x in the input volume belonging to channel c. The convolutional layer may also apply a nonlinear function, e.g. $tanh()$ or $max(0, x)$, on each value in the output volume, or allow for further customization of hyperparameters, like stride and zero-padding.

The **sub-sampling layer** is usually inserted between two convolutioanl layers and is employed to reduce the data size of the various feature maps, while retaining the relevant features. It is usually implemented as either a *max-pooling* or *mean-pooling* function, which substitutes an input submatrix with its maximum value or its mean, respectively.

The **classification** stage is then implemented by means of a classical Fully-connected Network (FCN), i.e. a series of fully-connected layers, sometimes followed by a final normalization operator.

The **fully-connected layer** is composed by J simple neurons (referred to as *perceptrons*), whose output values, for each $j \in J$, consists of a weighted linear combination of the previous neurons:

$$o_j = \sum_{i=0}^{I} (w_{i,j} \cdot x_i) + b_j \qquad (2)$$

where $w_{i,j}$ represent the weights, x_i the neurons from the previous layer, and b_j is an optional bias. The last linear layer will be constituted by a number of neurons equal to the number of classification classes.

When present, the **normalization operator** is in charge of computing the affinity between the input from the last linear layer and the classification classes as a percentage. It is usually implemented by means of a $LogSoftMax$ operator σ:

$$\sigma_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \quad for\ j = 1, \ldots, K \qquad (3)$$

where x_j is the output of the last linear layer. Such operator enforces the K values of the output to lie in range [0, 1] and to sum up to 1, expressing therefore the likelihood of the input to belong to each of the classification classes.

B. Streaming Stencil Time-step

Introduced in [3], a Streaming Stencil Time-step (SST) is a dataflow accelerator of a single ISL time-step – i.e. the outermost loop iteration – that is replicated several times to create a long chain of replicas that constitute the FPGA-based ISL accelerator proposed. Due to the resemblance of convolutions to single iterations (time-steps) of an ISL, we exploited the methodology of the work in [3] to derive the proposed dataflow accelerator of CNNs. It is therefore worth of interest to provide some details on the SST, as they can be useful to better understand the methodology prosed in this paper.

In an SST, as shown in Figure 2, we can identify a *memory system* and a *computing system*. The *computing system* consists of a collection of modules in charge of the actual computation, that are fed with the required data from the *memory system*. The *memory system* is instead composed of chains of modules, called *filters*, interconnected via FIFO channels, where each chain is responsible for storage and forwarding of data of one of the array involved in the computation. A chain receives as input a single data stream – the elements of the array referred – and the various filters represents the different accesses to that specific array that are needed to update a point during the ISL computation. A filter reads any existing element from it preceding FIFO and forward it to the subsequent one within the chain (if any). When the requested data arrives then, such filter, using its filtering equation, sends the data to the computing system allowing for output production. The structure of each chain of filters allows for concurrent accesses to the same array while ensuring that the data is read only once from the off-chip memory, optimizing memory resource consumption on the FPGA and realizing *full buffering*[1].

[1]Data is stored on the on-chip memory only when needed, i.e. until all the computation depending on it have completed, and is then discarded.

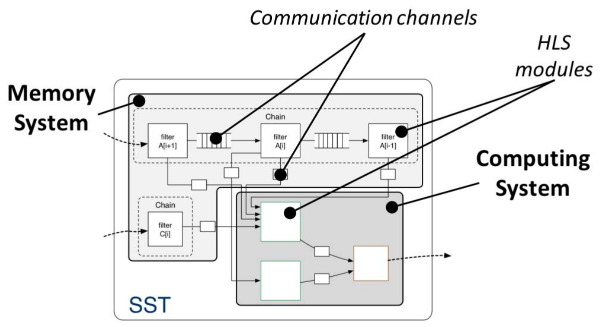

Fig. 2. Generic structure of an SST.

III. FPGA-BASED ACCELERATION METHODOLOGY

We now detail the proposed dataflow architecture for FPGA-based acceleration of CNNs, by describing how each layer is implemented (as a separate module) and how a complete network is constructed.

A. Convolutional and Sub-sampling Layer Architecture

Since they share similar memory access patterns, the convolutional and sub-sampling layers are here described together (with focus on the convolutional layer), as they mostly use the same memory and design optimizations. The convolutional layer has I_p input channel ports and O_p output channels ports, with $I_p \leq I_{FM}$ (input feature maps) and $O_p \leq O_{FM}$ (output feature maps). These interfaces can be used to receive/transmit multiple feature maps by interleaving the different feature maps over the same port, and are the parameters used to instantiate the *memory structure* and the *computation core*, which performs the convolutions and combine them. By acting on the number of instantiated I_p and O_p, the designer can perform a trade-off between parallelism and resource consumption. In particular, the number of output ports O_p represents the number of output feature maps written in parallel by the layer, while the number of input ports I_p gives the number of input feature maps accessed concurrently. A representation of the convolutional layer is shown in Figure 4. The *memory structure*'s architecture is based on the *memory system* of an SST, as described before in Section II-B. The values of the (potentially interleaved) feature maps are transferred into I_p chains of *filters* connected by FIFOs, where each filter redirect its input to the subsequent FIFO within the chain and to the computation core. Such organization of the memory structure allow the computation core, once the memory pipeline is filled, to receive an entire window for each clock cycle. Scalability of the memory structure with respect to the data size can be achieved via a memory/bandwidth trade-off, explained in [3]. As the convolutional layer may also present a *stride* parameter, indicating that the convolution window does not slide one pixel at a time, but skips some coordinates, the filtering conditions of the filters may need to be adapted to account for this eventuality.

To implement the *computation core* we need to specify a set of parameters related to the layer structure and the desired level of parallelism. Along with the number of input and output ports I_p and O_p, the core expects W_k H_k to be specified, as W_k and H_k represents respectively the width and height of the convolution window. The filters of the *memory structure*, as well as the computation core, have been implemented by means of Xilinx Vivado HLS. The pseudocode of the computation core is presented in Algorithm 1. The core has O_p output ports and $I_p \times (H_k \times W_k)$ input ports, all implemented using the *Axi4Stream* protocol. As described in Algorithm 1, the input is taken I_p feature maps at a time and then copied on a completely partitioned buffer. The *PIPELINE* directive is applied to all the internal loops, including also the input/output operations. The pipeline initiation interval is then computed as:

Algorithm 1 Pseudocode of the computation core of a convolutional layer

```
foreach $(x, y) \in Coordinates$ do
    $outputs \leftarrow biases$
    for $i = 0$ to $I_{FM}$ step $I_p$ do
        $buf \leftarrow I_p\ windows$
        $buf \leftarrow buf \cdot weights$
        $outputs \leftarrow outputs + reduce(buf)$
    end for
    send $outputs$ on $O_p$ ports
end for
```

$$Pipeline\ II = \max(\frac{O_{FM}}{O_p}, \frac{I_{FM}}{I_p}) \tag{4}$$

In the current implementation, the weights used to perform the convolutions are hardcoded in on-chip memory. The results of the multiplications are then added using a tree adder (the function *reduce*) and accumulated to the partial results. Such tree adder is used to perform the addition in parallel and therefore also decrease the pipeline depth. Once the needed convolutions are performed, the accumulated values are sent by the core as output on the output ports.

B. Fully-connected Layer Architecture

The structure of a fully-connected layer is actually similar to the structure of a convolutional layer, as indeed it can be considered a 1×1 convolution. Therefore, a fully-connected layer implementation can be done by using the same code and considerations made in Section III-A. However, the unitary window size makes the filters sequences of the *memory structure* not needed. Moreover, due to the very high number of input and output channels, a completely parallelized layer would utilize too much DSPs, exceeding very easily the physical limits of the FPGA. Thus, we decided to implement a fully-connected layer as the computation core of a single-input-port/single-output-port convolutional layer, where, for each input value, all the 1×1 convolutions of all the output feature maps are performed in the same clock cycle, while the output values are then produced sequentially after all the inputs have been processed.

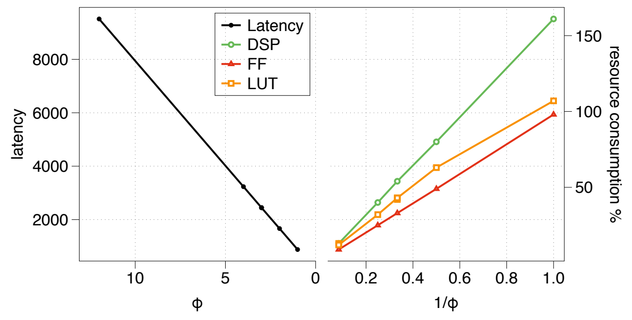

Fig. 3. Latency and Resource consumption figures of the computation core belonging to the first convolutional layer of the CNN for the CIFAR-10 dataset, in relation to $\phi = \max(\frac{O_{FM}}{O_p}, \frac{I_{FM}}{I_p})$. Notice that BRAM occupation is here not reported as for computation cores is always 0.

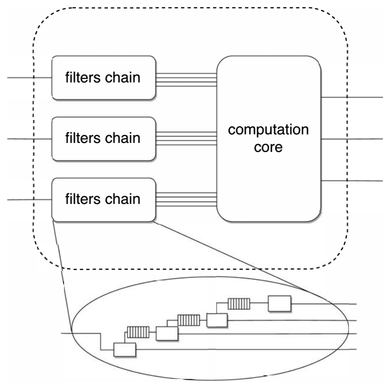

Fig. 4. Representation of a convolutional layer. The design is based on the one of an SST, shown in Figure 2.

C. CNN Accelerator Design

The design of the entire CNN accelerator requires the designer to perform a trade-off, for each layer, between level of parallelism and resource occupation. Indeed, the designer should explore the different level of parallelism and resource occupation to make the implementation feasible – i.e. it does not exceed the available resources – and attempt at minimizing the maximum latency among the different computation cores of the layers. Indeed, as will be shown in the experimental evaluation, the proposed CNN accelerator acts like a high-level pipeline, where the different layers are the "stages". At steady state, all the different layers of the CNN are concurrently active and performing some computation. Such characteristics can be exploited by feeding the CNN with batches of multiple images, resulting in an improvement in throughput. Therefore, minimizing the maximum latency among the computation cores of the layers will result in the improvement of the maximum achievable throughput of such pipeline.

Although in this work we did not perform any automated Design Space Exploration (DSE), we analyzed resource consumption and latency of each computation core and found a strong correlation between these variables and the maximum port/feature-map ratio $\phi = \max(\frac{O_{FM}}{O_p}, \frac{I_{FM}}{I_p})$. Indeed, we discovered that we can linearly relate the growth of resource requirements to $\frac{1}{\phi}$, while latency is linearly related to ϕ. Latency and resource consumptions with respect to ϕ and $1/\phi$ of a computation core (also considering different permutations of I_p and O_p that results in the same ϕ) are shown in Figure 3. The trend is the same for all computation cores that we designed.

We can therefore estimate latency and resource occupation of each computation core for each combination of I_p and O_p. The process consists of synthesizing via HLS just 2 cores that represents an arbitrary permutation of I_p and O_p, and get the resource estimation by Vivado HLS. We can then simply compute the slope σ of the underlying linear function, to estimate every other point of such function:

$$l = \sigma(\phi - \phi^0) + l^0 \tag{5}$$

to estimate the latency, where ϕ^0 and the latency l^0 are the values retrieved from either one of the 2 cores synthesized. To instead estimate the resource occupation of FFs, DSPs, and LUTs:

$$r_i = \sigma_i(\frac{1}{\phi} - \frac{1}{\phi^0}) + r_i^0 \qquad \forall i \in R = \{\text{FF,DSP,LUT}\} \tag{6}$$

The impact on resource occupation of the *memory structure* of the convolutional and sub-sampling layers is computed following the approach described in [3, 13]. For each filters chain, resource occupation is given by the sizes of the FIFOs and the occupation of the HLS cores implementing the filters. Also, each filters chain in a layer is of the same structure, and therefore of the same size m_c. In fact, memory occupation m_n of a layer $n \in N$ can be estimated as:

$$m_n = m_c^n \cdot I_p^n \tag{7}$$

We remark that our experimental evaluation was conducted without performing any automatic DSE, and instead we just determined empirically the levels of parallelization and therefore the number of I/O ports, essentially to find a layout that fits on the FPGA chip. Future work will address the automation of the DSE phase using the considerations made above. It will be also taken into account the possibility of splitting the layers into multiple parallel sub-modules (where each sub-module computes a sub-set of the output feature maps). Notice that sub-sampling modules can be directly splitted into parallel versions as they are applied independently to every feature map. We can therefore create as many sub-sampling modules as there are output ports of the previous convolutional layer, reducing the design complexity.

All the distinct layer cores are then connected sequentially through their I/O ports.

IV. Evaluation

A. Eperimental Setup

The two tested CNNs have been implemented, using Vivado HLS and Vivado IPI 2016.3, on a Xilinx VC707 board, mounting a Virtex-7 (xc7vx485t) FPGA, and running them at a 100MHz frequency. To perform the tests, each implemented network has been integrated with a soft-core processor (Microblaze) coupled with an Axi-Timer, an Axi4-Interconnect and a DMA to handle the communication from/to the CNN cores.

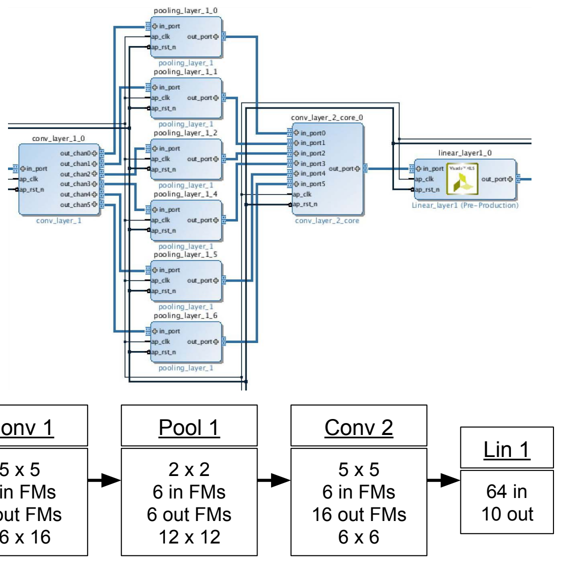

Fig. 5. Block design of the CNN for the USPS dataset, along with the hyperparameters of the CNN.

TABLE I
Performance and Power efficiency results

	GFLOPS	Power Efficiency (GFLOPS/W)	Image Latency (ms)	Images/s
USPS	5.2	0.25	0.0058	172414
CIFAR-10	28.4	1.19	0.128	7809
[14] (CIFAR-10)	-	-	-	2318

B. Experimental Results

Two CNNs have been implemented and tested, both with single floating point precision. The first network is composed of 4 different layers, and is trained and tested with images from the USPS dataset, composed of handwritten digits (16x16 grayscale images) from the U.S. Postal Service. From this point forward we will refer to this network as simply *USPS*. The network structure and block design are shown in Figure 5. In our design we were able to parallelize completely the first convolutional and sub-sampling layers. The second CNN is composed of 6 layers, and it is trained and tested against images from the CIFAR-10[1] database (32x32 RGB images). This network will be referred to as simply *CIFAR-10*. The network structure and block design can be seen in Figure 6. In this case, we implemented every convolutional layer as single-input-port/single-output-port. We tested both CNNs against an increasingly high batch of images, from 1 to 1000, to assess the validity of our claims about the high-level pipeline. Indeed, Figure 7 shows that the average time to process an image diminishes when the number of images per batch increases, until it reaches convergence to approximatively 5.8 μs for *USPS* and 128.1 μs for *CIFAR-10*. It is interesting to notice that, in both cases, convergence is reached approximatively when the size of the batch of images becomes greater than the total number of layers of the CNN. Table I then shows the performance and power efficiency figures. Notice that the measurements of performance are done taking also into account data transfer, as it is interleaved with computation. We compare our *CIFAR-10* implementation with the work by Microsoft Research in [14], as it is the only work that deals with the acceleration of a CNN on FPGA (an Altera Stratix V D5) for the same dataset. With the proposed approach, we were able to yield 3.36x better performance than [14].

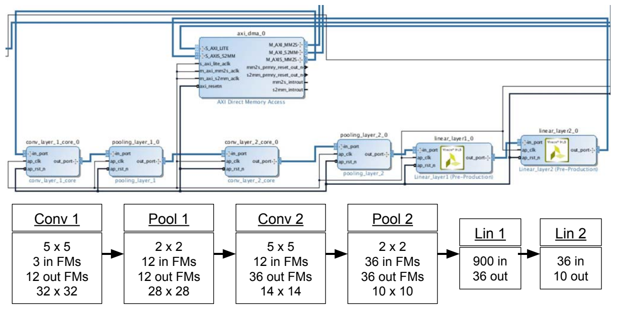

Fig. 6. Block design of the CNN for the CIFAR-10 dataset. The different hyperparameters of the layers are also reported.

We aimed, with this evaluation, at assessing the validity of the proposed methodology and proving the effectiveness of the high-level pipeline. We were able to perform our tests on relatively small networks and without performing proper DSE. We will investigate the validity of our approach on larger CNNs in future work, and perform a proper comparison with the literature.

V. Conclusions and Future Work

We presented in this paper a methodology to design an FPGA-based dataflow accelerator for CNNs, building upon previous work on the acceleration of ISL[3]. Following the proposed methodology, the designer can perform a trade-off between parallelization – and therefore performance – and resource occupation of the accelerator, by scaling each layer from single-input-port/single-output-port to fully parallel. We analyzed the impact of the design choices and provided a formula to estimate latency and resource occupation in relation to the degree of parallelism.

The approach has been evaluated with two CNN designs, on respectively the USPS dataset, and the CIFAR-10 dataset. We were able to show that the high-level pipeline implemented by the accelerator can be exploited to improve the throughput when the CNN is fed with batches of images, converging to a fixed value when the pipeline is completely filled.

Future work will explore different topics. Indeed, we will focus on the automation of the DSE (exploiting the estimation formulae of Section III-C)), and perform tests on larger and more popular CNN models like VGG[17] or AlexNet[10]. We will also focus on optimizing the design itself, by exploring further design optimization for the fully-connected layers and better exploit the available off-chip memory bandwidth. Moreover, we are investigating the possibility of integrating a completely automated design flow with existing industry-standard frameworks.

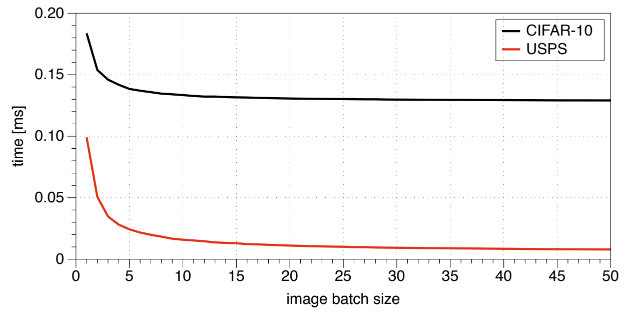

Fig. 7. Average time to process an image as the number of the images in the batch increases. For both CNNs, the high-level pipeline is exploited to improve the throughput. To improve readability we show the results only up to a batch of 50 images, as convergence is already reached.

Acknowledgments

This work is partially supported by the EXTRA project, that runs from September 2015 till August 2018 and receives funding from the EU Horizon 2020 research and innovation program under grant No 671653.

References

[1] "CIFAR-10," accessed: 22th of April 2017. [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[2] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, "A programmable parallel accelerator for learning and classification," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 273–284.

[3] R. Cattaneo, G. Natale, C. Sicignano, D. Sciuto, and M. D. Santambrogio, "On how to accelerate iterative stencil loops: a scalable streaming-based approach," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, p. 53, 2016.

[4] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 247–257.

[5] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[6] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, S. Dasgupta and D. Mcallester, Eds., vol. 28, no. 3. JMLR Workshop and Conference Proceedings, May 2013, pp. 1337–1345. [Online]. Available: http://jmlr.org/proceedings/papers/v28/coates13.pdf

[7] A. Dundar, J. Jin, V. Gokhale, B. Krishnamurthy, A. Canziani, B. Martini, and E. Culurciello, "Accelerating deep neural networks on mobile processor with embedded programmable logic," in *Neural information processing systems conference (NIPS)*, 2013.

[8] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 2009, pp. 32–37.

[9] S. Ji, W. Xu, M. Yang, and K. Yu, "3D convolutional neural networks for human action recognition," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 1, pp. 221–231, 2013.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[11] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio, "An empirical evaluation of deep architectures on problems with many factors of variation," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, pp. 473–480.

[12] Y. LeCun, K. Kavukcuoglu, and C. Farabet, "Convolutional networks and applications in vision," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, May 2010, pp. 253–256.

[13] G. Natale, G. Stramondo, P. Bressana, R. Cattaneo, D. Sciuto, and M. D. Santambrogio, "A polyhedral model-based framework for dataflow implementation on fpga devices of iterative stencil loops," in *Proceedings of the 35th International Conference on Computer-Aided Design*. ACM, 2016, p. 77.

[14] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, no. 11, 2015.

[15] M. Peemen, A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, Oct 2013, pp. 13–19.

[16] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A massively parallel coprocessor for convolutional neural networks," in *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*. IEEE, 2009, pp. 53–60.

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[18] K. Yu, "Large-scale deep learning at baidu," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. ACM, 2013, pp. 2211–2212.

[19] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *Proceedings of the 35th International Conference on Computer-Aided Design*, ser. ICCAD '16. New York, NY, USA: ACM, 2016, pp. 12:1–12:8. [Online]. Available: http://doi.acm.org/10.1145/2966986.2967011

[20] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.