Exploring Data Size to Run Convolutional Neural Networks in Low Density FPGAs

Ana Gonçalves[1], Tiago Peres[1], and Mário Véstias[2](✉)

[1] ISEL, Instituto Politécnico de Lisboa, Lisbon, Portugal
[2] INESC-ID, ISEL, Instituto Politécnico de Lisboa, Lisbon, Portugal
mvestias@deetc.isel.ipl.pt

Abstract. Convolutional Neural Networks (CNNs) obtain very good results in several computer vision applications at the cost of high computational and memory requirements. Therefore, CNN typically run on high performance platforms. However, CNNs can be very useful in embedded systems and its execution right next to the source of data has many advantages, like avoiding the need for data communication and real-time decisions turning these systems into smart sensors. In this paper, we explore data quantization for fast CNN inference in low density FPGAs. We redesign LiteCNN, an architecture for real-time inference of large CNN in low density FPGAs, to support hybrid quantization. We study the impact of quantization over the area, performance and accuracy of LiteCNN. LiteCNN with improved quantization of activations and weights improves the best state of the art results for CNN inference in low density FPGAs. With our proposal, it is possible to infer an image in AlexNet in 7.4 ms in a ZYNQ7020 and in 14.8 ms in a ZYNQ7010 with 3% accuracy degradation. Other delay versus accuracy ratios were identified permitting the designer to choose the most appropriate.

Keywords: Convolutional Neural Network · FPGA · Data quantization

1 Introduction

A CNN consists of several layers in a dataflow structure starting with the input image until the final layer that outputs a classification result. Each layer receives IFMs (Input Feature Map) from the previous and generates OFMs (Output Feature Map) to the next. The main and most common layers are: convolutional, fully connected and pooling.

Convolutional layers are the main modeling blocks of a CNN. For each IFM a 2D convolutional kernel is applied to generate a partial output map. The partial maps and a bias are accumulated to generate an OFM.

The set of 2D kernels form a 3D kernel. Each 3D kernel slides over the IFMs and the convolutions produce an OFM. CNNs consider several kernels at each convolutional layer and so the same number of OFM are produced at each layer.

C. Hochberger et al. (Eds.): ARC 2019, LNCS 11444, pp. 387–401, 2019.
https://doi.org/10.1007/978-3-030-17227-5_27

Convolutional layers may be followed by pooling layers to sub-sample the OFMs to achieve translation invariance and over-fitting. Pooling reduces the size of the feature map by merging neighbor neurons into a single neuron using functions like max or average pooling.

The last layers are usually the fully connected (FC). In a FC layer each neuron is connected to all neurons of the previous layer. The last FC layer outputs the classification probabilities. A nonlinear activation function is applied on every neuron. A common function recently adopted for its simplicity and effectiveness is the Rectified Linear Unit (ReLU) that calculates max(0, activation value).

Several CNNs has been developed with different number and type of layers, and number of kernels. One of the first was LeNet [3] with a total of 60K weights. The model was applied for digit classification with small images. Later, a much larger CNN, AlexNet [10], won the ImageNet Challenge. It consists of five convolutional layers plus three fully connected layers. Different number of kernels with different sizes are applied at each layer with a total of 61M weights requiring a 724 MACC (Multiply-ACCumulate) operations to process images of size $224 \times 224 \times 3$. Other CNN models have followed, like VGG-16 [12], GoogleNet [13] and ResNet [8].

Executing a CNN model (inference) can be done on the same platform used to train it or in an embedded system with strict performance, memory and energy constraints. In a vast set of applications, it is advantageous or necessary to have the inference process near the data input sensor so that important information can be extracted at the image sensor instead of sending the information to the cloud and wait for the answer. Also, in systems where the communication latency and data violations are undesirable, like autonomous vehicles, local processing at the sensor is also desirable.

A common feature of these CNN models is the high number of weights and operations. Due to the limited performance and memory of many embedded platforms it is very important to find architectural solutions to run large CNN inferences in low cost embedded platforms. One approach to achieve such implementations is to reduce the type and size of data without compromising the network accuracy. Size reduction reduces the complexity of arithmetic units and the memory requirements to store feature maps and weights.

In this paper, the focus is on the optimization of LiteCNN [16] for running inference of large CNNs in low density FPGAs (Field-Programmable Gate Arrays) using data size reduction.

The following has been considered for the optimization of LiteCNN:

– Lite-CNN only supports 8 bits dynamic fixed-point. An extended framework based on Caffe [9] and Ristretto [7] was developed to explore other fixed-point sizes;
– LiteCNN modifications are proposed to support generic fixed-point sizes;
– A performance model for LiteCNN was developed to allow design space exploration;

– Tradeoffs among performance, area and accuracy were obtained allowing the designer to choose the most appropriate LiteCNN configuration for a particular CNN model and accuracy.

The paper is organized as follows. Section 2 describes the related work on FPGA implementations of CNNs and optimization methods based on data size reduction. Section 3 describes the flow used to explore data size reduction of CNNs. Section 4 describes the LiteCNN architecture, the modifications necessary to support other data sizes and the performance model. Section 5 describes the results on inference accuracy and area/performance of LiteCNN running well-known CNNs and compare them to previous works. Section 6 concludes the paper.

2 Related Work

Common general processing units achieve only a few hundred GFLOPs with low power efficiency. This performance is scarce for cloud computing and the energy consumption is too high for smart embedded computing. GPUs (Graphics Processing Units) and dedicated processors (e.g. Tensor Processing Unit - TPU) offer dozens of TOPs and are therefore appropriate for cloud computing.

FPGAs are increasingly being used for CNN inference for its high energy efficiency, since it can be reconfigured to adapt to each CNN model.

The first FPGA implementations of CNNs considered small networks [1,2]. A larger CNN was implemented in [19] but only for the convolutional layers.

A few authors considered low density FPGAs as the target device. In [14] small CNNs are implemented in a ZYNQ XC7Z020 with a performance of 13 GOPs with 16 bit fixed-point data. In [5] the same FPGA is used to implemented big CNN models, like VGG16, with data represented with 8 bits achieving performances of 84 GOPs.

In [4] the authors implemented a pipelined architecture in a ZYNQ XC7Z020 with data represented with 16-bit fixed point. The architecture achieves 76 GOPs with high energy efficiency.

Previous implementations on low density FPGAs have performances below 100 GOPs. Previous works [6,11] show that dynamic fixed-point with 8 bits guarantee similar accuracies compared to those obtained with 32-bit floating point representations. In [17] hybrid quantization schema is proposed with different quantizations for different layers targeting edge computing. To deal with this hybrid quantization, the authors propose a pipeline structure with a layer at each pipeline level. The problem is that a pipeline structure requires enough memory to store intermediate feature maps and so it is not adequate for low density FPGAs with scarce memory resources.

Datawidth reduction is essential to implement CNN in target platforms with low on-chip memory and low resources. In this work we consider data bitwidths that can vary between layers and between activations and weights and study the impact of this hybrid quantization over the inference delay, accuracy and hardware resources.

The extended LiteCNN architecture with support for hybrid quantization proposed in this work is able to achieve several hundred GOPs in a low cost FPGA, like the ZYNQ7020, improve the inference delays of the original LiteCNN and of previous works and achieve high area and performance efficiencies. With LiteCNN, we have determined the tradeoffs between area, performance and CNN accuracy. Our solution improves the CNN inference delays of previous works in low density FPGAs with similar network accuracies.

3 Framework for Bitwidth Optimization

We have developed a framework based on Caffe [9] and Ristretto [7] to explore the bitwidth of both activations and weights. Ristretto determines the number of bits to represent data enough to guarantee a maximum error in the precision of the network specified by the user. To explore particular bitwidth sizes, we established a design flow with the following steps:

– The network is initially trained with single precision floating-point;
– Ristretto is applied to the trained network with different precision errors, generating solutions with different datawidths;
– From the results, we extract the fixed-point quantifications from each solution;
– From these values, we create a generic linear model of the quantification parameters (fractional and integer parts of fixed-point quantification);
– From this model we generate architectures with the required number of bits and train them to determine their accuracy.

Usually for a given target hardware architecture some data size configurations are more efficient than others in terms of area/performance. So, we want to explore these more efficient solutions in terms of network accuracy. This design flow permit us to determine the network accuracy for specific data bitwidths.

4 LiteCNN Architecture - 8 Bits

4.1 LiteCNN Architecture

The Lite-CNN architecture consists of a cluster of processing elements (PE) to calculate dot-products, a memory buffer to store on-chip the initial image and the OFMs, one module to send activations and two modules to send and to receive weights to/from the PEs (see Fig. 1).

The architecture executes layers one at a time. The execution of convolutional and fully connected layers work the same because we transform the 3D convolutions in linear dot-products identical to those used in FC layers, to be explained above. Layers are executed the following way:

– Before starting the execution of a layer, the architecture is configured for the specific characteristics of the layer. It also specifies if there is a pooling layer at the output of the feature maps being calculated;

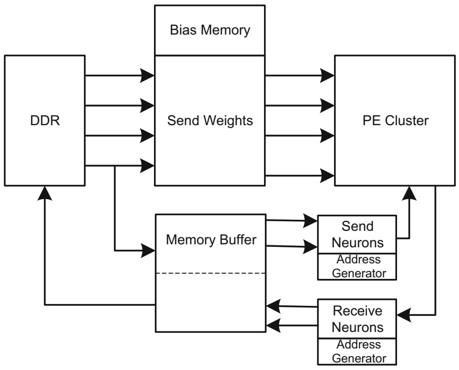

Fig. 1. Block diagram of the Lite-CNN architecture

- The input image and the intermediate feature maps are stored on-chip. Since the layers are executed one at a time, the buffer memory only has to be enough to store the IFM and OFM of any layer;
- For the first convolutional layer, the image is loaded from external memory. For the others, the IFM is already in on-chip memory. At the same time, kernels are read from external memory and sent to the PEs. Besides the weights, the kernel includes the bias value which is stored in the bias memory. Each PE receives one kernel. So, each PE calculates the activations associated with one OFM;
- The initial image or intermediate feature maps in the on-chip memory are broadcasted to all PEs;
- After each calculation of a complete dot product associated with a kernel, all PEs send the output activations back to the receive neurons module that adds the bias and stores the result in the on-chip memory to be used by the next layer. If the layer is followed by pooling, this module saves the activations in a local memory and wait for the other members of the pooling window;
- The process repeats until finishing the convolution between the image and the kernels. After that, the next kernels are loaded from memory and the process repeats until running all kernels of a layer.

The process allows overlapping of kernel transfer and kernel processing. While the PEs process their kernels, in case the local memory is enough to store two different kernels, the next kernels are loaded at the same time. This is fundamental in the fully connected layers where the number of computations is the same as the number of weights.

Also, in case the on-chip memory is not enough to store the whole image and the OFM (usually the first layer is the one that requires more on-chip memory), the image is cut into pieces which are convolved separately.

The PE cluster contains a set of PEs. Each PE (see Fig. 2) has a local memory to store kernels and arithmetic units to calculate the dot product.

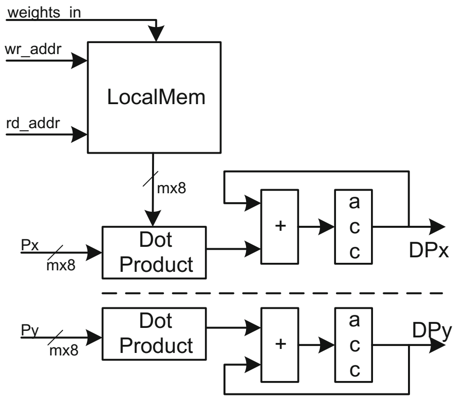

Fig. 2. Architecture of the processing elements

Each PE stores a different kernel and so is responsible for calculating the activations of the output feature map associated with the kernel. This way multiple output feature maps are calculated in parallel. Also, in convolutional layers, the same kernel is applied to different blocks of the IFM and produce different neurons of its OFM. The number of output neurons to be processed in parallel in each PE is configurable. For example, to calculate two activations in parallel it receives two input activations from the feature memory in parallel. This mechanism permits to explore the intra-output parallelism (fully connected layers do not use intra-output parallelism). Finally, weights and activations are stored in groups, that is, multiple weights and activations are read in parallel in a single memory access (e.g., with 8-bit data, a 64 memory word contains eight neurons or weights) permitting to explore dot-product parallelism.

The block *sendWeights* is configured to send kernels to the PE cluster. The block receives data from direct memory access (DMA) units that retrieve data from external memory and send it to the PEs in order. It includes a bias memory to store the bias associated with each kernel.

The *sendNeurons* and *receiveNeurons* blocks are responsible for broadcasting activations from the feature memory to the PEs and receive dot product results from the PEs, respectively. The send neurons module includes a configurable address generator. The receive neurons module implements the pooling layer in a centralized manner.

Previous works use dedicated units to calculate 2D convolutions. The problem is that the method becomes inefficient since the same units have to run different window sizes and are used only for convolutional layers. Lite-CNN transforms 3D convolutions into a long dot product to become independent of the window size. Also, this way, both convolutional and FC layers are executed the same way by the same arithmetic core units.

Pixels of the initial image, activations of feature maps and weights of kernels are stored in order (z, x, y) (see Fig. 3).

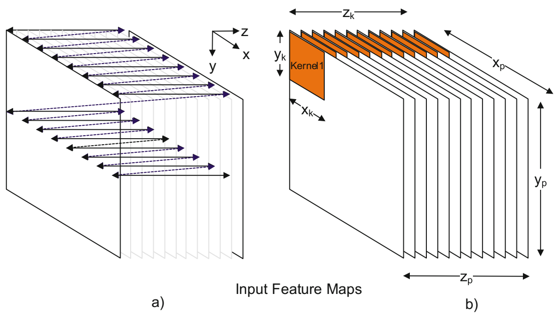

Fig. 3. Reading mode of images, feature maps and weights

Each neuron of an OFM is calculated as a dot product between the 3D kernel of size $x_k \times y_k \times z_k$ and the correspondent activations of the IFM of size $x_p \times y_p \times z_p$ (see Fig. 3b), where z_p is the number of IFMs. The weights of kernel are all read sequentially from memory since they are already ordered. The activations are also read in sequence from memory but after $x_k \times z_k$ activations it has to jump to the next y_k adding an offset to the address of the input feature memory being read. For a layer without stride nor followed by pooling, the offset is $x_p \times z_p$. Formally, the dot product to calculate each step of the convolution is given by:

$$DP_{conv} = \sum_{i=0}^{i=y_k-1} \sum_{j=0}^{j=x_k z_k-1} W_{ix_k z_k+j} \times P_{startAddr+ix_p z_p+j} \tag{1}$$

where *startAddr* is the address of the first neuron of the block of the IFM being convolved. We use this operation to convolve a kernel with the set of IFMs sliding the 3D kernel along the feature maps. In this process, if a layer is followed by pooling, the output neurons of the pooling set are calculated in sequence and only the final neuron is stored in the OFM buffer. The advantage of our method is that it is independent of the shapes of kernels and weights and layer type. WE just have to configure the address generator properly for each layer.

LiteCNN also implements a method to reduce the number of multiplications by half [16] leading to a considerable reduction in the hardware resources required to implement a convolutional or fully connected layers. Also, the intra-output parallelism used during convolutional layers can be used to batch IFM to be sent to FC layers. This version of LiteCNN supports two parallel lines of computation. Each of these lines can be used to process one of the batched IFM for the FC layers, that is, it supports a batch of two.

We have extended LiteCNN with two modifications to support data sizes different from *activation* $\times$ *weight* $= 8 \times 8$. Since we cannot afford having a pipelined datapath with dedicated implementations of each layer, due to low memory resources, we keep the generic layer implementation that is configurable to support each particular layer and extended it to support different data sizes.

In those cases where all layers use the same sizes (e.g., 16×16, 5×5, 8×2), the processing units are configured exactly to execute operations with this size. We kept the memory data bus with 64 bits and so the number of parallel units depend on the size of activations and weights (64/size).

When layers have different data sizes, we store data in their original sizes, but core units are implemented to support the execution of the bigger operands. Therefore, data with smaller dimension are extended to the size of data with the biggest dimension. For example, consider two different representations in the same CNN - 8×4 and 8×2 - the arithmetic units are implemented for 8×4 and 8×2 data is extended to 8×4 to be executed. In this extended version of LiteCNN, cores support multiply-accumulations of data with upto two different data representations whose sizes are configured initially. For example, it can be configured to execute layers with size 8×4 and 8×2, or 8×8 and 8×2. Extending LiteCNN to obtain architecture configurations that support the execution of more than two different data sizes is straightforward but was not considered in this paper.

With this architectural solution using layers with different data representations has no computational advantage, since the number of operations is the same as using the same data sizes, but the data is read and stored from/to memory in their original sizes. So, the method permits to take advantage of using reduced weight sizes to reduce the time to transfer activations and weights from memory. Designing generic arithmetic units was left for future research.

The second modification of the PEs has to do with the method to reduce the number of multiplications. When both activations and weights have the same size, the method is used. Otherwise, the method is less efficient since the multiplications have the size of the bigger parameter (e.g. if 8×4, the size of the multiplications is 9×9). In these cases we adopted and extended for other dimensions the method proposed in [15].

4.2 Performance Model of LiteCNN

The performance model provides an estimate of the inference execution time of a CNN network on the LiteCNN architecture. The model determines the time to process each layer.

Considering convolutional layers, the time to transfer all kernels depends on the number of kernels, *nKernel*, the size of kernels, *kernelSize*, the number of bits used to represent weights, *nBit* and the memory bandwidth, BW. The total number of bytes, *tByte*, transferred in each convolutional layer is given by Eq. 2.

$$tByte = nKernel \times kernelSize \times \frac{nBit}{8} \tag{2}$$

The number of cycles to execute a convolutional layer, *conCycle*, is given by Eq. 3.

$$convCycle = \left\lceil \frac{nKernel}{nCore} \right\rceil \times \frac{nConv \times kernelSize}{nMAC} \tag{3}$$

where *nCore* is the number of processing elements, *nConv* is the number of 3D convolutions and *nMAC* is the number of parallel multiply-accumulations of each PE (intra-output parallelism). From these two equations, the total execution time, *convExec* depends on the local memory capacities. If local memories of PEs have enough space to store two kernels, than communication and processing of kernels can overlap, otherwise, they must be serialized. Considering an operating frequency, *freq* de execution time is given by Eq. 5.

$$convExec = \frac{tByte}{BW} + \frac{convCycle}{freq} \qquad \text{without overlap} \tag{4}$$

$$convExec = max(\frac{tByte}{BW}, \frac{convCycle}{freq}) \qquad \text{with overlap} \tag{5}$$

For the totally connected layers, the equation to determine the number of bytes to transfer all kernels is the same as Eq. 2. The equation to determine the number of cycles to process the layer is given by:

$$fcCycle = \left\lceil \frac{nKernel}{nCore} \right\rceil \times \frac{kernelSize}{nMAC} \times nParallel \tag{6}$$

Since in the fully connected layers there is no intra-output parallelism, only one line of parallel MACs of the PE is used. Given the number of intra-output parallel processing lines, *nParallel*, the number of processing cycles is multiplied by this value.

The total execution time of FC layers is similar to 5.

$$fcExec = \frac{tByte}{BW} + \frac{fcCycle}{freq} \qquad \text{without overlap} \tag{7}$$

$$fcExec = max(\frac{tByte}{BW}, \frac{fcCycle}{freq}) \qquad \text{with overlap} \tag{8}$$

The total execution of a CNN inference in LiteCNN is the sum of the time to transfer the image to FPGA ($\frac{imageSize(bytes)}{BW}$) plus the time to process each layer. Between layers there is configuration time of the architecture done by the ARM processor of ZYNQ. We have checked the accuracy of the model from the results of LiteCNN 8×8 running AlexNet. The delay obtained with the model is about 1% lower (17.44 ms) against (17.63 ms) of the implementation.

5 Results

We have tested LiteCNN with data size reduction with one small network - LeNet5 - one medium size CNN - Cifar10-full - and one large CNN - AlexNet. Cifar10-full is a network with three convolutional layers and one fully connected layer used to classify images from the CIFAR-10 dataset containing 32×32 color images. All LiteCNN architectures were implemented with Vivado 2017.3 in the ZedBoard with a ZYNQ XC7Z020 and run at 200 MHz.

Table 1. Area occupation for different data size configurations of Lite-CNN

Layer x	4×4	5×5	6×6	16×16	8×8	8×8	8×8	8×4	8×4	8×2	2×8
Layer y	4×4	5×5	6×6	16×16	8×8	8×4	8×2	8×4	8×2	8×2	2×2
PEs	64	64	64	32	64	64	64	43	43	40	38
MACC	32	24	20	16	16	16	16	32	32	64	64
LUT	47477	44922	44895	45098	44418	46624	46832	45824	47842	45641	45430
DSP	220	220	220	220	220	220	220	220	220	220	220
BRAM	130	130	130	132	130	130	130	115	115	111	111
Peak GOPs	819	614	512	205	410	410	410	563	563	1024	972

For each CNN we found the relation between delay and accuracy when implemented in LiteCNN. Since LiteCNN is configurable in terms of processing elements, to facilitate the comparison of architectures with different data size configurations, we implemented all architectures with similar areas by changing the number of processing elements (see area results in Table 1).

Table 1 gives the number of PEs and the number of MACC in each PE for a particular implementation of LiteCNN (layer x and layer y lines indicate the size of the operands supported in each implementation). A line with the peak performance was also included. With layers configured with 4×4 data sizes the architecture has a peak performance of 819 GOPs and configured with 8×2 it has over 1 TOPs of peak performance.

Considering these implementations (with similar areas), we have determined the accuracy (top-1) of the networks (LeNet, Cifar10-full, AlexNet) for different data size configurations and the delay. To avoid long synthesis times of all architectures, we used the performance model to determine the delay (the performance model was verified for the original LiteCNN).

LeNet is a small network and is used for simple number recognition. Therefore, it has high accuracy and executes fast compared to the other larger networks. We have considered data of convolutional layers with the same size and varied the size between convolutional and FC layers (see Fig. 4).

Each architecture configuration is specified by the bitwidth of activations (the same for all layers, specified as A:size) and the bitwidth of weights (can be different across all layers, specified as W:size.. if the same for all layers or W: followed by all sizes of each layer when different).

The fastest solution is obtained with configuration (A:8; W:4442). The reason is that with 2-bit weights in the FC layers, it reduces the high data communication delay of weights in FC layers. In fact, we observe that the increase in delay is related to size of weights in fully connected layers. In terms of accuracy, it increases with the datawidth of activations and weights but the delay increases more than linearly with the increase in accuracy.

Cifar10-Full has 3 convolutional layers and 1 FC layer. The accuracies of Cifar10-Full are lower than that of LeNet because the classification problem is more complex (see Fig. 5).

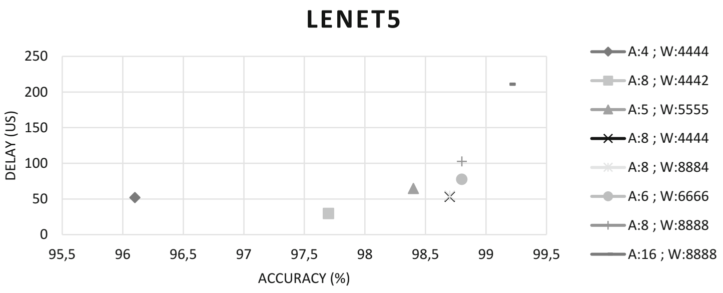

Fig. 4. Accuracy versus delay for different configurations of LiteCNN running LeNet

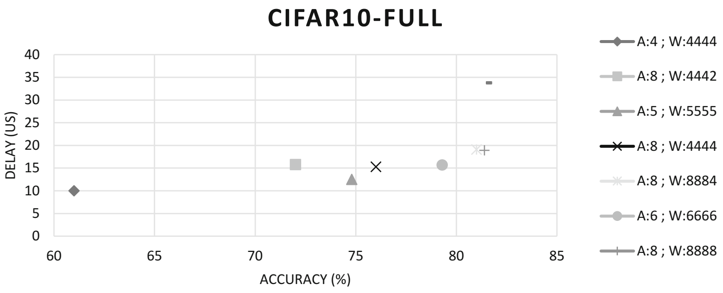

Fig. 5. Accuracy versus delay for different configurations of LiteCNN running Cifar10-Full

The results for Cifar10-Full are slightly different than those for LeNet. The FC layers of Cifar10 are not the bottleneck since the size and number of kernels are close to those of the convolutional layers. Therefore, those configurations with a smaller number of bits for FC weights are not necessarily better; configurations from A:8 W:4444 to A:8 W:8888 have a small variation in delay (around 3 us) for 10% variation in accuracy.

AlexNet is larger and requires more bits to represent data in order to maintain acceptable accuracies. In this case, we considered an hybrid size of weights, that is, two possible sizes of weights in different layers keeping activations with the same size for all layers. The results were compared with state of the art implementations in the ZYNQ board with a low density SoC FPGA - ZYNQ7020. We have also mapped these different configurations of LiteCNN in a ZYNQ7010. As far as we know, this is the first attempt to implement a large CNN in the smallest SoC FPGA of the ZYNQ family from Xilinx (see Table 2).

The results reveal the importance of determining the right bitwidth of data. Moving from the configuration with the highest accuracy (A:16; W:16..) to a configuration with almost the same accuracy (A:16; W:8..) the delay improves 43%. The biggest improvements occur when there is a reduction in the size of the weights. Reducing the activations has a lower impact on the delay with a higher impact on the accuracy.

Table 2. Performance comparison of LiteCNN with previous works in low density FPGAs ZYNQ7020 and ZYNQ7010 SoC FPGAs

Work	Format	Freq (MHz)	Latency (ms)	Acc.
ZYNQ 7020				
[18]	A:16; W:16..	100	71,75	[(a)]
[14]	A:16; W:16..	125	52,4	[(a)]
[4]	A:16; W:16..	200	16,7[(b)]	[(a)]
LiteCNN	A:16; W:16..	200	33,8	55,6
	A:16; W:8..		19,4	55,5
	A:8; W:8..		17,4	54,4
	A:8; W:82222228		7,4	52,7
	A:4; W:82222228		6,6	49,5
	A:2; W:82222228		5,7	46,5
ZYNQ 7010				
LiteCNN	A:8; W:8..	200	24,8	54,4
	A:8; W:82222228		14,8	52,7
	A:4; W:82222228		12,2	49,5
	A:2; W:82222228		8,3	46,5

[(a)]Authors assume accuracy close to that obtained with floating-point - 55,9%
[(b)]With pruning and image batch

Compared to previous works, the proposed architecture improves the delay in more than 50%, except when compared with [4]. However, in this case, the proposed solution uses weight pruning and image batch, which are not considered in our proposal.

With LiteCNN we could map AlexNet in the smallest SoC FPGA from Xilinx - ZYNQ7010 - in a ZYBO board. As expected, inference delays are higher because it has much less resources (less PEs) and since the available on-chip RAM is not enough to hold the image and the first OFM, the image has to be halved and processed separately. The impact in the delay is higher when we reduce the size of the weights since in this case the computation times of the convolutional layers relative to the FC layers increases and the ZINQ7010 implementation has less PEs to calculate convolutional layers. For example, considering configuration (A:8; W:8..) the delay increases 1.4×, while for configuration (A:8; W:82222228) it increases 2×. However, notably, it can run AlexNet in real-time (30 fps).

To better understand the impact of size reduction of activations and weights on the inference delay, we have determined the time to execute convolutional layers and the time to execute FC layers (see Fig. 6).

The execution time of FC layers is higher than that of convolutional layers. The execution time of FC layers is dominated by the communication of weights from external memory. This fact degrades the average GOPs. Reducing the size of

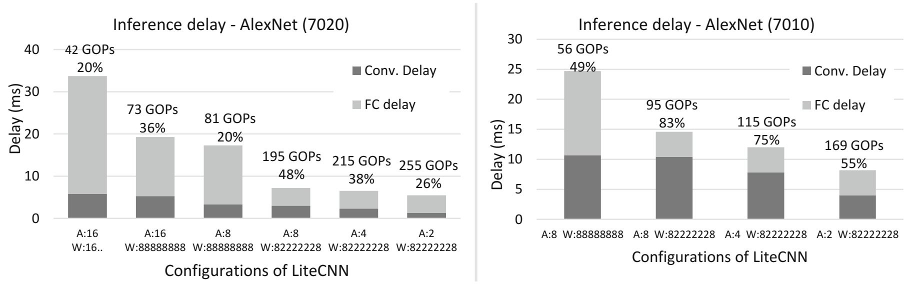

Fig. 6. Execution time of convolutional and FC layers for different configurations of LiteCNN running AlexNet

FC weights improves the real GOPs of the architecture. The real GOPs improves when LiteCNN is mapped on ZYNQ7010. In this case, the execution time of FC layers is about the same (the memory bandwidth is the same in both FPGAs) and the execution time of convolutional layers increase. So, the implementation in ZYNQ7010 is more efficient.

6 Conclusions

In this work we have developed a framework to explore the design space of bitwidth of activations and weights. LiteCNN was extended to support the execution of layers with different data widths.

The extended LiteCNN with configurable bitwidths improves the performance/area efficiency with a small impact over the inference accuracy of the CNN. This is fundamental for embedded systems with low resources.

We have also observed that weight size reduction has more effect on architecture optimization than activation size reduction since it not only permits to increase the performance/area ratio of the architecture but also reduces the time to transmit FC weights, the performance bottleneck in the execution of CNN models with large FC layers.

We are now studying in more detail the smallest size formats and how to compensate for the accuracy loss by changing the CNN model. We have also started to complement data size reduction with data reduction using techniques like pruning.

Acknowledgment. This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2019 and was also supported by project IPL/IDI&CA/2018/LiteCNN/ISEL through Instituto Politécnico de Lisboa.

References

1. Chakradhar, S., Sankaradas, M., Jakkula, V., Cadambi, S.: A dynamically configurable coprocessor for convolutional neural networks. SIGARCH Comput. Archit. News **38**(3), 247–257 (2010). https://doi.org/10.1145/1816038.1815993
2. Chen, Y., et al.: DaDianNao: a machine-learning supercomputer. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 609–622, December 2014. https://doi.org/10.1109/MICRO.2014.58
3. Cun, Y.L., et al.: Handwritten digit recognition: applications of neural network chips and automatic learning. IEEE Commun. Mag. **27**(11), 41–46 (1989). https://doi.org/10.1109/35.41400
4. Gong, L., Wang, C., Li, X., Chen, H., Zhou, X.: MALOC: a fully pipelined FPGA accelerator for convolutional neural networks with all layers mapped on chip. IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. **37**(11), 2601–2612 (2018). https://doi.org/10.1109/TCAD.2018.2857078
5. Guo, K., et al.: Angel-Eye: a complete design flow for mapping CNN onto embedded FPGA. IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. **37**(1), 35–47 (2018). https://doi.org/10.1109/TCAD.2017.2705069
6. Gysel, P., Motamedi, M., Ghiasi, S.: Hardware-oriented approximation of convolutional neural networks. In: Proceedings of the 4th International Conference on Learning Representations (2016)
7. Gysel, P., Pimentel, J., Motamedi, M., Ghiasi, S.: Ristretto: a framework for empirical study of resource-efficient inference in convolutional neural networks. IEEE Trans. Neural Netw. Learn. Syst. **29**(11), 5784–5789 (2018). https://doi.org/10.1109/TNNLS.2018.2808319
8. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, pp. 770–778, June 2016. https://doi.org/10.1109/CVPR.2016.90
9. Jia, Y., et al.: Caffe: convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093 (2014)
10. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: Proceedings of the 25th International Conference on Neural Information Processing Systems, NIPS 2012, vol. 1, pp. 1097–1105. Curran Associates Inc., USA (2012)
11. Ma, Y., Suda, N., Cao, Y., Seo, J., Vrudhula, S.: Scalable and modularized RTL compilation of convolutional neural networks onto FPGA. In: 2016 26th International Conference on Field Programmable Logic and Applications, FPL, pp. 1–8, August 2016. https://doi.org/10.1109/FPL.2016.7577356
12. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: Proceedings of the 3rd International Conference on Learning Representations (2015)
13. Szegedy, C., et al.: Going deeper with convolutions. In: 2015 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, pp. 1–9, June 2015. https://doi.org/10.1109/CVPR.2015.7298594
14. Venieris, S.I., Bouganis, C.: fpgaConvNet: mapping regular and irregular convolutional neural networks on FPGAs. IEEE Trans. Neural Netw. Learn. Syst. 1–17 (2018). https://doi.org/10.1109/TNNLS.2018.2844093
15. Véstias, M., Duarte, R.P., de Sousa, J.T., Neto, H.: Parallel dot-products for deep learning on FPGA. In: 2017 27th International Conference on Field Programmable Logic and Applications, FPL, pp. 1–4, September 2017. https://doi.org/10.23919/FPL.2017.8056863

16. Véstias, M., Duarte, R.P., de Sousa, J.T., Neto, H.: Lite-CNN: a high-performance architecture to execute CNNs in low density FPGAs. In: Proceedings of the 28th International Conference on Field Programmable Logic and Applications (2018)
17. Wang, J., Lou, Q., Zhang, X., Zhu, C., Lin, Y., Chen., D.: A design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA. In: 28th International Conference on Field-Programmable Logic and Applications (2018)
18. Wang, Y., Xu, J., Han, Y., Li, H., Li, X.: DeepBurning: automatic generation of fpga-based learning accelerators for the neural network family. In: 2016 53rd ACM/EDAC/IEEE Design Automation Conference, DAC, pp. 1–6, June 2016. https://doi.org/10.1145/2897937.2898002
19. 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, FPGA 2015, pp. 161–170. ACM, New York (2015). https://doi.org/10.1145/2684746.2689060