Generating Efficient FPGA-based CNN Accelerators from High-Level Descriptions

Nermine Ali, Jean-Marc Philippe, Benoit Tain, Philippe Coussy

Generating Efficient FPGA-based CNN Accelerators from High-Level Descriptions

Nermine Ali[1*], Jean-Marc Philippe[1], Benoit Tain[1] and Philippe Coussy[2]

[1*]Université Paris-Saclay, CEA List, Palaiseau, 91120, France.
[2]Lab-STICC, Université de Bretagne-Sud, Lorient, 56321, France.

*Corresponding author(s). E-mail(s): nermine.ali@cea.fr;
Contributing authors: jean-marc.philippe@cea.fr;
benoit.tain@cea.fr; philippe.coussy@univ-ubs.fr;

Abstract

The wide landscape of memory-hungry and compute-intensive Convolutional Neural Networks (CNNs) is quickly changing. CNNs are continuously evolving by introducing new layers or optimization strategies to either improve accuracy, reduce memory and computational needs or both. Moving such algorithms to on-device enables smarter edge products. However, hardware designers find this constant evolution hard to master, which keeps CNN accelerators one step behind. More approaches are using reconfigurable hardware, such as FPGAs, to design customized inference accelerators that are more suited to the newly-emerging CNN algorithms. Moreover, high-level design techniques, such as High-Level Synthesis (HLS), are adopted to address the time-consuming RTL-based design and the design space exploration problems. HLS allows generating RTL source code from high-level descriptions. This paper presents a hardware accelerator generation framework targeting FPGAs that relies on two steps. The first step characterizes the input CNN and produces hardware-aware metrics. The second step exploits the generated metrics to produce an optimized C-HLS source code for each layer of the input CNN, then it uses an HLS tool to generate a synthesizable RTL representation of the inference accelerator. The main goal of this approach is to reduce the gap between the evolving CNNs and the hardware accelerators, thus reducing design time of new systems.

Keywords: Convolutional Neural Networks, Design Space Exploration, High Level Synthesis, Hardware Accelerators, FPGA

1 Introduction

Convolutional Neural Networks (CNNs) [1] are bio-inspired algorithms, precisely, their topology is inspired by the hierarchical structure of neurons in the visual cortex. CNNs are composed of neurons that are organized in many computational layers. Generally, the first layers in CNNs rely on convolutional filters that extract features from an input. Those features will be processed by the last layers, known as fully connected layers, to classify the input. CNNs are one of the leading technologies applied for computer vision tasks, such as image classification, object detection and recognition. They acquire their knowledge in a supervised manner in a learning phase by using labeled data, and infer results in the inference phase, which is the main interest in this work.

CNN layers are based on computational operations throughout the network to classify the input (Figure 1). Some operations are based on convolutions in which weighted filters extract features from an input feature map. Convolutions are often followed by activation functions (e.g. Rectified Linear Unit - ReLU) that introduce non-linearity into the neural network. Other operations, known as pooling, reduce the number of parameters (i.e. weights) as well as the size of the activations map but maintain robust features only. The last layers, usually fully connected, perform the classification process.

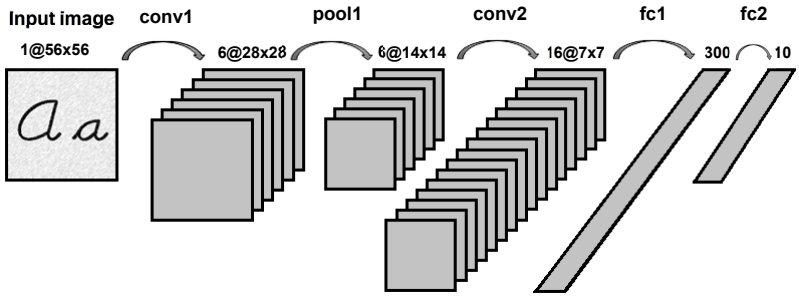

Fig. 1 Example of a CNN topology ([1]).

Usually, the accuracy of such algorithm is evaluated based on the top-1 and top-5 scores that a CNN can achieve on a dataset, such as the ImageNet Large Scale Visual Recognition Challenge [2]. CNNs are continuously evolving by exploiting various techniques to either improve accuracy or reduce memory and computational needs or both. Such techniques include leveraging depth and the spacial aspects, introducing new types of layers (e.g. depth-wise layers [3]). High-level optimizations are also exploited to reduce bit-precision of weights and activations by applying quantization, or to increase sparsity by pruning the CNN. These optimizations are applied to move these algorithms from cloud to edge devices to enable more real-time data processing closer to the source of the data, and avoid increased latency and performance degradation. This means that CNN accelerators are needed. Therefore, myriads

of CNN hardware accelerators are designed while trying to leverage various optimizations techniques to achieve an efficiency/flexibility compromise. However, the algorithmic evolution is hard to follow by hardware architects, which increases the gap between the application and the hardware accelerators. In addition, designing hardware accelerator is time-consuming and requires an advanced hardware expertise, especially if manually designed using RTL.

As a first step to reduce this gap, [4] proposes a characterization step for CNNs to gain a thorough understanding of the application. The proposed approach extracts target-agnostic but hardware-aware metrics to derive analysis, configurations and/or mapping strategies that can be of use for an accelerator implementation. A second step to reduce the gap is to automate the hardware generation and to employ high-level design techniques, such as High-Level Synthesis (HLS), to improve design productivity [5]. The extracted metrics in the first step can be used to optimize the accelerator generation that is based on HLS.

In [6], a hardware generation approach targeting FPGAs based on two phases has been proposed. The first step allows to deeply understand the CNN by characterizing its behavior and extracting relevant metrics. The second step, exploits HLS to generate an RTL representation of the hardware accelerator. The main goal of this approach is to efficiently design hardware accelerators for the inference phase while considering the quickly changing landscape of CNNs.

This paper extends the work in [6] and makes the following contributions:

- evaluation of the ability of the approach to perform quantization-aware hardware generation. RTL implementation results of the evaluation CNN quantized in 4 bits and 16 bits are presented respectively;
- comparison of the 4-bit and the 16-bit implementations to the 8-bit implementation resource-wise and performance-wise;
- evaluation of the approach on a state-of-the-art CNN, MobileNet-V2;
- exploitation of the characterization results of MobileNet-V2 to generate an optimized 8-bit RTL implementation.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 introduces the proposed hardware accelerator generation framework with details on the characterization part. Section 4 shows the results related to the execution of the framework on a small CNN, while in Section 5, a state-of-the-art CNN is used to evaluate the proposed approach. Finally, Section 6 concludes the paper and introduces future works.

2 Related Work

Reaching both high performance and high energy efficiency is a complex task when dealing with deep learning applications. For this purpose, different architectural and design approaches were proposed to obtain efficient CNN hardware accelerators. The approaches exhibit different tradeoffs between performance and flexibility. For example, fixed structures optimized for feature

extraction operations [7] sacrifice the flexibility to maximize the performance, while programmable architectures (either homogeneous [8] or heterogeneous [9]) allow to execute several CNN topologies, using their DPS-like execution models. Some architectural proposals rely on specific features exploiting the CNN algorithm itself (for example the degree of sparsity) or introduce hardware-aware techniques (such as pruning [10] or compression [11]) to reduce the energy consumption of the system.

The internal spatial structure of FPGA makes them interesting targets for the design of deep learning hardware accelerators. The resulting parallelism offers interesting opportunities for implementing accelerator structure suited to the dataflow nature of deep learning algorithms. Moreover, fine-grain reconfiguration of FPGAs allows both designing accelerators matching the topology of the CNN, and sizing computing and memory resources with required data precision. Several tools implementing different approaches are proposed by FPGA vendors or researchers to accelerate deep learning algorithms on FPGAS. For example, [12] proposes tiling techniques to improve computation of a pipelined architecture while [13] introduces a FPGA accelerator targeting sparse CNNs.

New design approaches and methodologies to design CNN accelerators need to be investigated to reduce the gap between changing deep learning algorithms and hardware architectures. These methodologies usually rely on techniques to abstract the hardware or to perform automatic exploration of the design space. Some frameworks directly generate a synthesizable high-level code using commercial HLS tools such as Catapult or Vivado-HLS, based on a network description. For example, the framework presented in [14] targets FPGAs by providing empirical estimations of hardware resources used by a hardware accelerator in a web interface. [15] focuses on setting optimized tiling parameters to tailor convolution accelerator templates written in HLS. This work proposes analytical models to estimate latency, and DSP and BRAM resources.

A more recent approach [16] introduces a framework enabling the generation of a hardware accelerator based on a specific CNN description. FP-DNN [17] is an approach based on RTL-HLS hybrid templates that generates CNN accelerators using proper HLS tools. FINN [18] deals with binarized neural networks. It takes as input a trained binary NN to generate a streaming computing architecture in the form of an optimized C++ description. hls4ml [19] is a modern approach that leverages quantization-aware training and pruning techniques optimize the NN topology and parameters before the generation of the accelerator. These techniques allow leveraging the fine-grain structure enabled by FPGAs or ASICs. Design space exploration is also used on existing architectures [20] or architectural templates [21]. While optimizing the hardware generation, these approaches do not allow to fully leverage the potentials of HLS tools since the considered accelerator descriptions are too tight to the actual architectures.

This paper proposes an intermediate approach between direct architecture generation and time-consuming design space exploration. It introduces a preliminary step in the form of an hardware-aware application characterization module that computes hints driving the high-level code generation step. The objective of the approach is to automatically design a CNN hardware accelerator by using a hardware generation framework based on a characterization phase and HLS techniques. The framework leverages the outputs of the CNN characterization step to help optimizing a high-level algorithmic representation of a CNN hardware accelerator. This approach is described in the next section.

3 Overview and Implementation of the Hardware Generation Framework

3.1 Overview of the approach

In the present work, a hardware generation methodology is proposed to automatically design an optimized accelerator from a CNN-based application description. This flow features a characterization phase, introduced in [4], that helps to apprehend the behavior of the input CNN to rapidly generate an optimized synthesizable RTL from high-level descriptions in a hardware generation phase. However, it is a challenging problem to produce an optimized RTL representation from application specifications.

Figure 2 sketches the overall flow of the proposed approach. The flow comprises two main phases: the characterization phase introduced in [4] followed by an HLS-based hardware accelerator generation phase. The flow takes as input a CNN application specification comprising a description file, which defines the CNN topology and the layers hyperparameters, and a database of images used in the testing/validation step of the CNN application design. The parameters of the input CNN can also be included in the description file if the file format supports it, such as *tflite* from the Tensorflow deep leaning framework or *onnx*, the de facto standard for describing CNNs.

The foremost step in the characterization phase extracts relevant metrics from the CNN description. Then, a specific module analyzes those metrics and combines them to deduce hints on mapping strategies to optimize data movement, or configurations of a target architecture, such as the number of processing elements and memory requirements. The next phase is a hardware generation step based on HLS, which leverages the resulted metrics to generate an optimized C-HLS source code. The generated source code is then fed to an HLS tool that will transform it into an optimized RTL representation. The flow is automated and does not require any manual tuning. However, the user might intervene in two cases: if the FPGA target needs to be changed due to resources usage constraints (the framework makes the user aware of it), or if the user desires to apply other optimizations that are not automatically

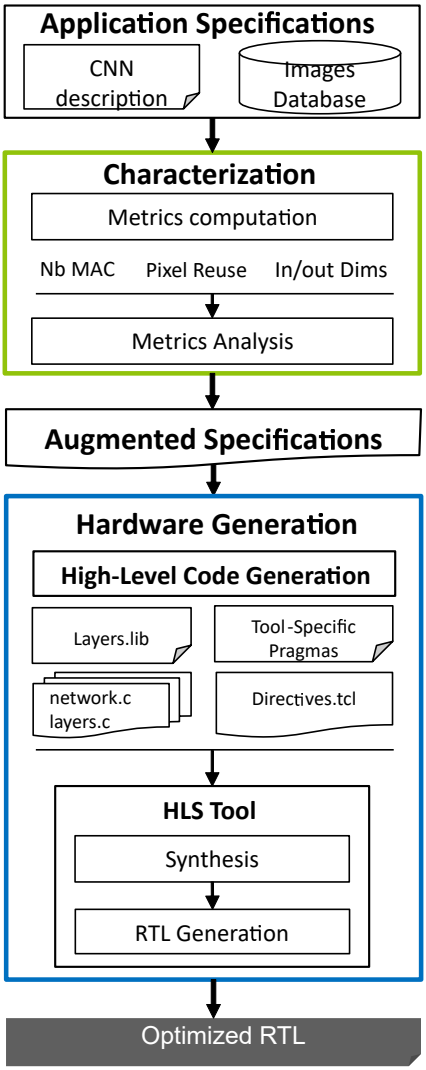

Fig. 2 Overview of the proposed exploration and accelerator generation framework flow.

applied by the framework. The following subsections provide more details on those two phases and their implementations.

3.2 Characterization and analysis phase

Figure 3 depicts the characterization phase and its different steps. A preliminary step, *Transformation*, parses the input CNN description and pulls out an internal representation (IR), which comprises layers types and the related hyperparameters. The IR is a list of objects, in which each object represents a layer with its configuration (i.e. type and kernel size) and the related metrics

that will later be computed, such as the dimensions of the input/output feature maps, pixel reuse percentage, the number of computational operations, etc. The *Transformation* step is implemented as a Python script that can be adapted to the input file format (e.g. *tflite*, *onnx*, etc.). Suitable parsing libraries are imported to the Python script to support the variety of the input file formats.

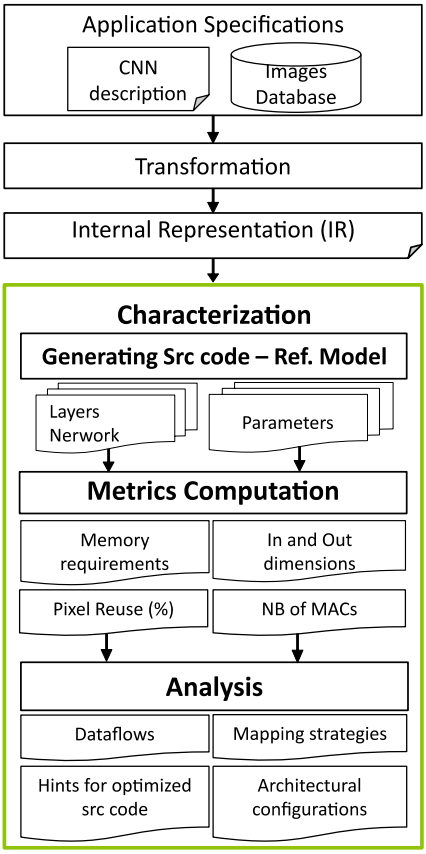

Fig. 3 Overview of the characterization phase.

The IR is used, first, for C code generation, second, to compute the metrics that characterize the behaviour of the CNN. For the first feature, a Python script generates a functional reference C code of the entire CNN to dynamically characterize the CNN and analyze the sparsity of the input feature maps using a test or validation dataset. This C-based implementation includes the C code of each layer composing the input CNN, such as convolutional (Conv.) layers, pooling (Pool.) layers and fully connected (FC) layers.

After generating the C source code, the metrics computation step takes place. In this latter, the C code is executed to analyze weights and feature

maps (using the database of images) on a per layer basis to obtain sparsity information, weights distribution as well as bit-precision. Another set of metrics is also computed (without the need to execute the generated C source code) for each layer composing the CNN by using a specific Python script in the characterization step, as detailed in here. This script uses the IR to compute target-agnostic/hardware-aware metrics for each layer. It computes the input and output dimensions of the feature maps throughout the CNN (if not available in the description file) by using the size of the input image and the layers configurations (i.e. kernels dimensions). The computation of the number of MAC (multiplication/accumulation) operations depends on the input/output feature maps dimensions, kernels dimensions, as well as on the type of the layer (i.e. standard/depthwise convolution, FC, etc.). The number of MAC of a standard convolution is calculated using Equation 1.

$$Nb_{std\ conv_{MAC}} = R \times C \times K_x \times K_y \times M \times N \tag{1}$$

Memory requirements include both weights and input activations, where each one is computed based on different dimensions. The required weights memory for each layer is computed by multiplying the dimensions of the convolutional kernels. The memory needs for activations are computed (for each layer) as the product of the input feature maps dimensions. The pixel reuse percentage is obtained based on the fact that a convolution is a sliding window over a feature map. Furthermore, a connection graph is generated by using *networkx* Python package, which allows to create and manipulate complex networks. It groups layers of the same type into one node. In addition, visual data is generated and provided to the user, such as charts and graphs.

The computed metrics are stored in a *.csv* file and are used to update the IR. The related script also checks if the input CNN is quantized by verifying the parameters bit-widths, which allows to set the right bit-precision-related optimizations in the Hardware Generation step.

Once the metrics are generated, they are then fed to an analysis process, which aims to use each metric or to combine a few of them to derive implementations strategies and provide hardware-based hints for an efficient implementation. For example, the sparsity analysis gives the number of zero-valued weights, which is a key information to help identify layers where memory requirements can be reduced by applying compression techniques. The pixel reuse percentage is an interesting metric that enables deriving several dataflows, which help in optimizing the energy efficiency and memory accesses of dataflow-based architectures. The input and output feature maps dimensions allow deriving tiling strategies. Furthermore, the pixel reuse percentage and the dimensions metrics can be combined and then employed to determine the achievable degree of parallelism and what type of reuse to prioritize, i.e. pixel reuse or weight reuse. On a higher level, algorithmic implementations can also be derived, such as loop tiling and reordering, which have a significant impact on the needed on-chip memory and volume of data transfer. Additionally, a connection graph is useful to reduce resource usage by assigning

identical layers or even similar ones, in terms of memory requirements and kernel dimensions, to the same hardware resources.

The analysis process is implemented as a Python script that verifies if the extracted metrics meet certain conditions (or rules), which are specific to the metrics being analyzed. For instance, if the average weights sparsity is more than 70%, a flag would be set to reduce computational needs by skipping zero-valued computations. In this work, no zero-skipping module is implemented. However, it is a perspective of this work. In addition, the analysis script combines certain metrics based on their types and potential impacts. For instance, the width/depth comparison and the pixel reuse metrics are analyzed together via a Python method presented in Listing 1, called *MetricCombine*, which illustrates only one use case. This method analyzes these two metrics for every layer in the CNN by taking four input parameters: *loopsOrder*, *width*, *depth* and *pixelReuse*. The *loopsOrder* parameter comprises the original loops order of the given layer. If the width exceeds the depth and the percentage of pixel reuse is greater than 50%, then the loops in a loop nest (e.g. in convolutional layers) should be reordered to encourage data reuse. Therefore, the loops corresponding to the width and height dimensions are moved closer to the computational part (kernel loops). Detailed examples can be found in sections 4 and 5.

```
# initial loops order [0, 1, 2, 3, 4, 5]
MetricCombine(loopsOrder, pixelReuse, width, depth):
    if width > depth and pixelReuse > 0.5: # pixel reuse in %
        # set new order
        newOrder = [2, 3, 0, 1, 4, 5]
        # change the loops order
        loopsOrder = [loopsOrder[i] for i in newOrder]
    return loopsOrder
```

Listing 1 Pseudo-code of the combination of the width/depth comparison and the pixel reuse metrics.

Herein, the analysis of the characterization results drives the C-HLS code generation for an HLS-based implementation, and sets the right tiling parameters as well as the right pragmas.

3.3 HLS-based accelerator generation phase

This phase relies on two main steps. The first step generates an optimized C-HLS code of the entire CNN by means of a library of layers of different types. It exploits the obtained metrics in the previous step (see section 3.2) to guide the C-HLS code generation of each layer by determining the loops order in a loop nest, the loop levels to tile as well as the suitable tiling parameters. Moreover, the analysis of those metrics allow setting HLS tool-specific pragmas to optimize the overall CNN accelerator performance-wise and/or resource-wise, and to obtain a hardware implementation with a right tradeoff between resource usage and latency. This tradeoff is currently obtained by optimizing each layer performance-wise and the whole CNN resource-wise. The optimized C-HLS code generation is performed by a specific Python script that takes into account the results of the characterization phase. For instance, the loops that require re-ordering are directly generated in the right order. Three separate files

are generated: *layers.c* comprising all the layers (written as C functions), and *network.c* including a *network()* function that calls all the layers, in addition to a *directives.tcl* file that includes the pragmas. The source code also includes loops and functions labels that will be used in the directives file (*directives.tcl*). This file includes the pragmas (unrolling, pipeling and allocation) that are applied to specific loops and functions in the C-HLS code.

The second step takes as input the generated optimized source code of the whole CNN as well as the directives file. This step leverages an HLS tool to transform the C-HLS code to an optimized synthesizable RTL implementation.

4 Evaluation of the Framework

The proposed methodology is validated using a sample evaluation CNN to produce an optimized accelerator. The framework presented in Figure 2 is instantiated using CNN description files from N2D2 (Neural Network Design and Deployment), an open source deep learning framework [22] that allows designing and deploying Deep Neural Network applications. N2D2 also includes quantization and pruning features to target embedded systems. Regarding the hardware generation phase, Vivado-HLS is used as an HLS tool that targets Xilinx FPGAs. The employed version is 2020.1.

The sample evaluation CNN, described in Figure 4, comprises six layers, which are: 3x3 and 5x5 2D-convolution, 3x3 and 5x5 pooling and 2 fully-connected layers. It was trained with N2D2 on the Caltech-101 dataset [23] to classify four classes of images (airplanes, car sides, faces, motorbikes). The accuracy on the validation database is 95.35%. This evaluation CNN was then quantized after training to *8*-bit integer format using a post-quantization technique provided in N2D2. The accuracy after quantization is 94.02%. Therefore, *int8* data format was employed for data and weights when generating the C-HLS source code.

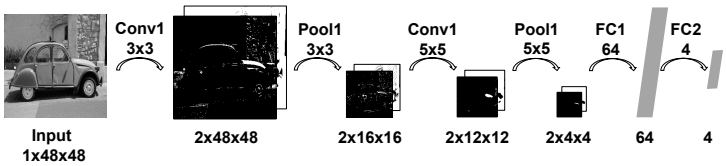

Fig. 4 Details of the layers of the evaluation CNN.

4.1 Applying the characterization step on the evaluation CNN

The evaluation CNN is first characterized layer by layer memory-wise and computational-wise. Table 1 shows the number of computing operations for each layer (*MAC* for convolutional and fully connected layers, *Ops* for pooling and softmax layers) as well as the memory needs. As it can be seen, the first

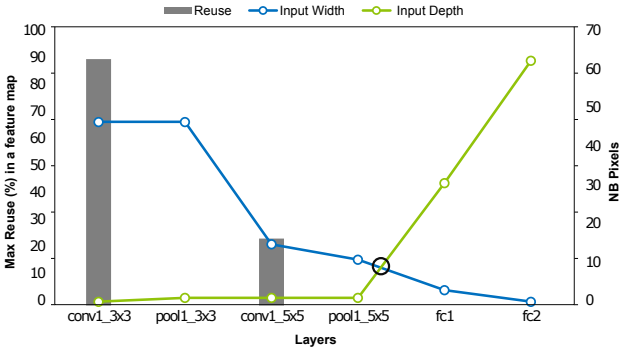

Fig. 5 Widths and depths comparison of the input feature maps in each layer of the evaluation CNN together with pixel reuse percentage.

layer is the most computationally intensive one and has the most data memory needs due to the size of the input image, which requires more processing than the remaining layers.

Additional metrics are also computed, such as the maximum pixel reuse and the width/depth comparison of input activations maps across the CNN. Those metrics are presented in Figure 5. One can see that the width shrinks throughout the CNN, since the size input feature maps decrease due to the applied convolutional and pooling operations. On the other hand, the depth increases, especially in fully connected layers, due the the significant number of filters. As for the pixel reuse percentage, it is high in the first layer, *conv1_3×3*, having a value of *90%* due to the large input size. This reuse percentage drops below *30%* in *conv1_5×5*, since the input feature map decrease across the CNN.

4.2 Generation of the hardware accelerator for the evaluation CNN

The metrics obtained from the *Characterization step* are used to drive the generation step of the hardware accelerator optimized for the evaluation CNN. The

Table 1 Computational complexity and memory requirements for different layers of the evaluation CNN.

Layers	MACs | Ops for pool & softmax	Parameters (B)	Data memory (B)
Input data	0	0	2304
conv1_3×3	41472	18	4608
pool1_3×3	4608	0	512
conv1_5×5	14400	100	288
pool1_5×5	288	0	32
fc1	2048	2048	64
fc2	256	256	4
softmax	4	0	4
Total	58176	2422	5512

different layers are separately implemented to evaluate different configurations depending on the position of the layer in the overall structure of the CNN. The different implementations use a frequency of 100MHz while targeting the Xilinx Zynq7000 xc7z030 FPGA.

4.2.1 Generation of the different convolutional layers

Listing 2 presents a common approach for the implementation of a convolutional layer. This standard implementation can be improved using the computed metrics. Depending on the metrics, two loop transformations (loop tiling and loop reordering) can be applied at relevant loop levels.

```
ifmap[N][(R-1)*S+K][(C-1)*S+K] //input maps
outfmap[M][R][C]   //output maps
weights[M][N][K][K]
l0: for(r=0; r<R; r++)    //output X
 l1: for(c=0; c<C; c++)   //output Y
  l2: for(m=0; m<M; m++)      //nb outputs
   l3: for(n=0; n<N; n++)     //nb channels
    l4: for(kx=0; kx<Kx; kx++)     // kernel X
     l5: for(ky=0; ky<Ky; ky++)    // kernel Y
       wx=weights[m][n][kx][ky]
       ix=ifmap[n][S*r+kx][S*c+ky]
       outfmap[m][r][c]+=wx*ix
```

Listing 2 Common approach to implement a convolutional layer given in the form of a pseudo-code.

4.2.2 Loop transformations

Loop transformations in loop nests are essential to optimize data movement and memory accesses, which improves the overall performance if the transformations are correctly applied. Such transformations include loops tiling, re-ordering, unrolling and pipelining.

Loop tiling is a commonly used loop transformation, especially in HLS approaches for CNNs [24] [15] [25]. It limits expensive memory accesses and transfers by encouraging data locality. Additionally, it improves performance if the tiling parameters are correctly set. When tiling loops, new loop levels are incorporated to the source code as presented in Listing 3 with various tiling parameters T. The number of added loops depends on the number of tiled loops in the loop nest. However, deciding the loop levels to tile as well as their tiling parameters is a challenging task. Therefore, characterization results come into play to help determining which loop level to tile as well as the tiling parameters. The metrics to consider are the pixel reuse percentage and the width/depth comparison. Combining these two metrics allows to identify the loops to be tiled. Furthermore, it allows to set threshold values for the tiling parameters to restrain the design space. Equation (2) shows the minimum and maximum threshold values for tiling parameters (Tr, Tc) of the outputs width and height (R, C), and (Tn, Tm) of the channels N and outputs M respectively. For instance, the minimum threshold value for the height R is set to Kx, which is the kernel width.

$$\begin{aligned} Kx \leq Tr \leq R \\ Ky \leq Tc \leq C \\ tn \leq Tn \leq N \\ tm \leq Tm \leq M \end{aligned} \tag{2}$$

For instance, layer *conv1_3×3*, the first layer in the CNN (see Figure 4), has the largest input and output feature maps. Figure 5 shows that this layer has the highest width (48) and the lowest number of channels (1). The large width explains the significant pixel reuse percentage of 92%, which is the highest among all the layers. Two actions could be taken after analyzing those two metrics: determining the loop level to tile and identifying the tiling parameters. Herein, loops *l0* and *l1*, which are the loops of the width and height of the output feature map, are suitable for tiling due to their large bounds. This allows to improve data locality and reuse by storing chunks of data on-chip and thus optimizing memory accesses. The tiled source code of the convolutional layer is shown in Listing 3.

```
ifmap[N][(R-1)*S+K][(C-1)*S+K] //input maps
outfmap[M][R][C]   //output maps
weights[M][N][K][K]
l0: for(r=0; r<R; r+=Tr)      //output X
 l1: for(c=0; c<C; c+=Tc)     //output Y
  l2: for(m=0; m<M; m++)      //nb outputs
   l3: for(n=0; n<N; n++)     //nb channels
    l0.1: for(tr=r; r<min(R,r+Tr); tr++)
     l1.1 for(tc=c; c<min(C,c+Tc); tc++)
      l4: for(kx=0; kx<Kx; kx++)      // kernel X
       l5: for(ky=0; ky<Ky; ky++)     // kernel Y
        wx=weights[m][n][kx][ky]
        ix=ifmap[n][S*tr+kx][S*tc+ky]
        outfmap[m][tr][tc]+=wx*ix
```

Listing 3 Implementation of a tiled convolutional layer in the form of a pseudo-code.

Similarly, loops *l0* and *l1* are the ones to be tiled in layers *pool1_3×3* and *conv1_5×5*, since the width of the feature maps in these layers is larger than the depth. To show the impact of tiling on resource usage and latency, various tiling parameters are studied on the different layers. Each layer is implemented separately (from C to place and route), while infrastructures such as memory and crossbars are omitted. As it can be seen in Figure 6, increasing the tiling value improves performance while having a small impact on hardware resources. Loops *l2* and *l3* are kept intact since the number of channels and the number of outputs are very small. Table 2 and Table 3 show the latency (in cycle) as well as the resource utilization (in percentage) of layers *conv1_5×5* and *pool1_5×5* respectively. As it can be seen, the applied tiling in those layers have a slight impact on performance and resource usage, which can be explained by the low pixel reuse percentage (see Figure 5), where layer *conv1_5×5* has a pixel reuse of 25% and the remaining layers have a pixel reuse of 0%.

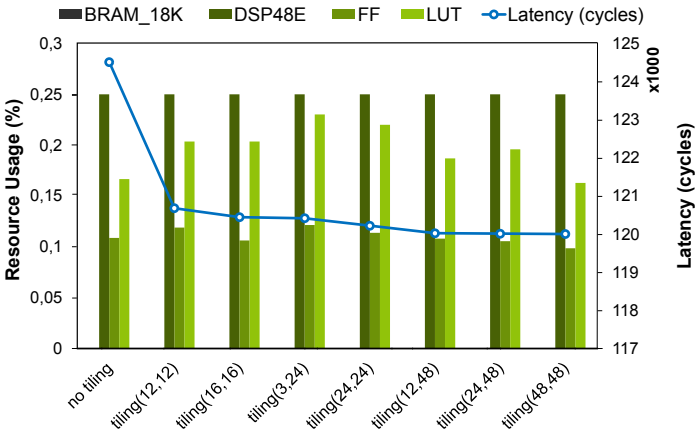

Fig. 6 Latency and resource usage of different implementations of the *conv1_3×3* layer using different tiling values.

Allowing a certain amount of data to be stored or available on-chip could also be achieved by correctly re-ordering the loops. The same rule applied to identify which loop level to tile could be applied to determine the loops order in a loop nest to optimize performance. The same loops that are tiled could be exchanged with other loop levels to reduce latency, since this improves data locality as well as memory accesses. The pixel reuse as well as the input and output dimensions of a layer help to identify the loops order. For instance,

Table 2 Resource usage percentages and latencies for *conv1_5×5* using different tiling values.

Tiling *(Tr, Tc)*	BRAM	DSP	FF	LUT	Latency (cycles)
(0, 0)	0	0.25	0.114	0.181	36602
(3, 3)	0	0.25	0.132	0.239	36586
(4, 4)	0	0.25	0.119	0.242	36494
(6, 6)	0	0.25	0.127	0.256	36414
(6, 12)	0	0.25	0.118	0.223	36350
(12, 12)	0	0.25	0.109	0.191	36341

Table 3 Resource usage percentages and latencies for *pool1_3×3* using different tiling values.

Tiling *(Tr, Tc)*	BRAM	DSP	FF	LUT	Latency (cycles)
(0, 0)	0	0	0.065	0.104	13858
(4, 4)	0	0	0.063	0.127	13674
(8, 8)	0	0	0.061	0.117	13470
(8, 16)	0	0	0.0.058	0.104	13390
(16, 16)	0	0	0.057	0.104	13381

swapping (*l0*, *l1*) with *l2*, in layer *conv1_3×3*, results in the same performance and resources as using (48, 48) as tiling parameters that are the maximum possible values. This can be explained by the fact that more data is available on-chip and close to the computational part, which consequently improves memory accesses and performance. This is also applicable on the remaining convolutional and pooling layers.

Unrolling and pipelining are also significant loop transformations that have a great impact on the generated RTL performance-wise and resources-wise. The implementation of those transformations differs from one HLS tool to another. In Vivado-HLS they are implemented as pragmas, also known as directives. More details are presented in the following subsection.

4.2.3 Applying transformations using pragmas

Unrolling and pipelining are loop optimizations related to performance. Applying such optimizations results in significant changes in the RTL. Loop unrolling a loop introduces spatial parallelism to reduce latency by allowing a concurrent execution of a number of iterations set via an unrolling factor N, which allows a full or partial unroll. In a full unroll mode, the latency of the loop is reduced to the latency of one iteration. Hardware-wise, unrolling creates necessary RTL copies of the loop that corresponds to the unrolling factor N, which implies an increased need for hardware resources. As for the piplining optimization, it allows a simultaneous execution of operations inside a function or a loop to improve the latency, more specifically to reduce the initiation interval. From a hardware perspective, this optimization also requires more hardware resources to allow this concurrent execution. Those two optimizations are applied as pragmas in the HLS tool.

Optimizing each layer performance-wise relies on applying transformations to the loop nest. Loop swapping, discussed in the previous subsection (section 4.2.2), resulted in a better performance compared to loop tiling. Therefore, layers will have loop levels (*l0*, *l1*) swapped with *l2*. In addition, pragmas will be applied on those layers to boost the performance further. This first set of optimizations will be called LSP, which stands for Loop Swapping and Pragmas. The framework takes into consideration the topology of the network to apply the pragmas. Typically, loops having very small bounds (i.e. layers having small dimensions), such as the channels loop (*l3*) and/or outputs loop (*l0*) as well as the kernel loops (*l4*, *l5*), are completely unrolled. This is the case in the first four layers of the evaluation CNN, where kernel loops and outputs loop are fully unrolled. Unrolling the outputs loop (*l0*) in fully-connected layers would result in a significant resource usage due to the high number of outputs, especially in *fc1* where the number of outputs is 64 (input of *fc2*). Therefore, only the two innermost loops are pipelined to obtain reasonable resources utilization and latency. Unrolling and pipelining are also applied in layers where no loop swapping is performed nor any other loop transformation. This is called $NLSP$ for short, where only pragmas are applied. Figure 7 presents the results of these two implementations, where $sol{-}0$ refers to $NLSP$ and

sol−1 refers to *LSP* (i.e. Vivado-HLS *solutions*). Only pragmas are applied in layers *fc1* and *fc2*, since loop swapping/tiling showed no enhancement in performance nor resource usage.

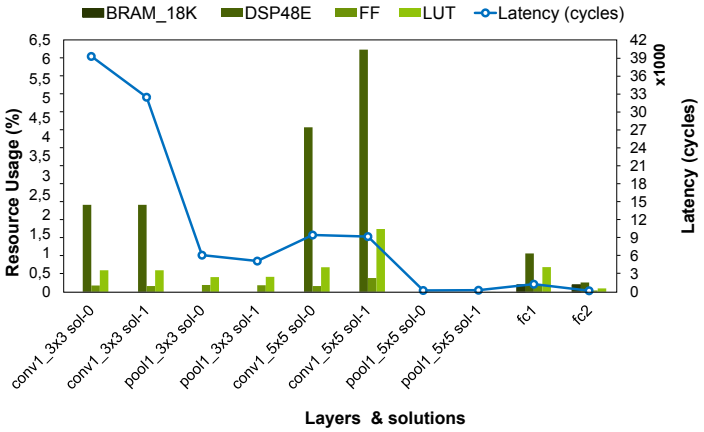

Fig. 7 Latency and resource usage of the optimized implementations of the layers of the evaluation CNN.

According to Figure 7, the *LSP* implementation of layer *conv1_3×3* is 17.34% faster than its *NLSP* implementation while requiring globally 0.34% less resources. As for the *LSP* implementation of *pool1_3×3*, it is faster than the *NLSP* but requiring 0.33% more resources. Similarly, the *LSP* implementation of *conv1_5×5* is 2.76% faster than the *NLSP* and has 38.7% more resource usage (BRAM and DSPs). The *NLSP* implementation of *pool1_5×5* is 12% faster and requires 50% more resource usage than *LSP* implementation. The improved latency in the *NLSP* implementation can be explained by the fact that the input/output feature maps have small dimensions, which means that the data reuse is low, and therefore data locality could not be well exploited.

Some pragmas are not an application of loop transformation, they are synthesis directives that directly impact the RTL to be generated. For instance, the Allocation pragma could be employed inside a region of code (e.g. a loop or a function) to control resource utilization. It allows to limit the number of RTL instances to reduce the required resources implementing functions, operations, etc. For example, a function called n times in a C code would have n RTL instances. This pragma allows to limit the RTL instances to a certain number and allows to use the same RTL instance for the same high-level functions. The use of this pragma will be demonstrated in Section 5.

4.3 Generation of the full hardware accelerator

The resulting optimized layers are then used to generate the whole hardware accelerator. The evaluation of the applied strategies was performed using

multiple implementations. In the first implementation, no optimization (loop swapping or pragmas (NLS)) is applied. Only loop swapping (LS) is applied in the second implementation, and finally, both loop swapping and pragmas (LSP) optimizations are applied for the third implementation. The results of these implementations are shown in Figure 8.

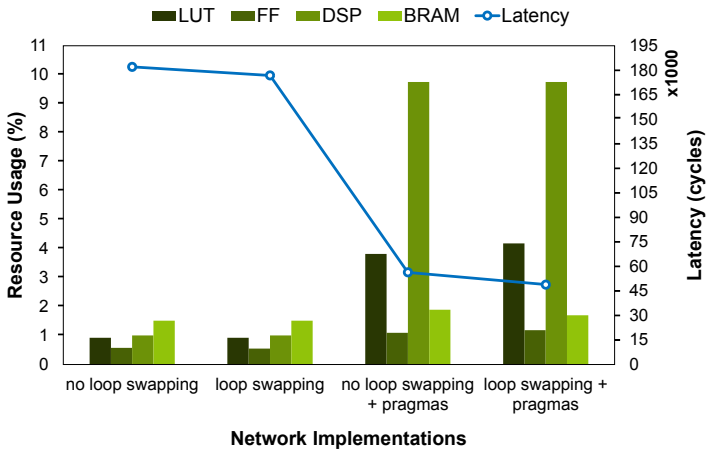

Fig. 8 Latency and resources utilization of each implementation of the CNN used for evaluation.

One can see in Figure 8 that the LS implementation is 3% faster than the NLS implementation. Moreover, it uses 0.68% less FPGA resources. The application of tool-specific pragmas optimizations to both NLS and LS implementations leads to 69% and 72% speed-up respectively, while requiring 76% more resources. The LSP implementation is 13% faster then the $NLSP$ implementation at the price of 1.55% additional resource utilization. The $NLSP$ and LSP implementations are respectively 69% and 73% faster than a standard implementation. Unfortunately, the higher performance comes at the price of additional area cost, implying that the chosen implementation strongly depends on the target design goal.

4.4 Hardware generation using arbitrary bitwidths

The proposed framework allows generating hardware accelerators for quantized CNNs, as stated earlier. To evaluate the behavior of the quantization-aware hardware generation, the evaluation CNN is quantized in 4 bits and 16 bits respectively. The accuracy is 91.95% after quantizing to 4-bit, and 94.11% after 16-bit quantization. Indeed, quantizing to 4-bit decreased the accuracy of the CNN by around 4%, which is still reasonable due to the significant advances in the quantization field. The non-optimized version of the evaluation CNN is then synthesized (place and route) for each used precision. Table 4 summarizes the results of these implementations and compares them with the

8-bit non-optimized implementation. As it can be seen, the resource utilization and the latency gradually increase with the increase of the number of used bits. The INT4 implementation is *0.11*% and *24.15*% faster compared to INT8 and INT16 respectively. Regarding resource utilization, the DSP usage is *1.0*% for all implementations. The BRAM usage is *25.16*% and *40.21*% lower than both INT8 and INT16. The same goes for FF utilization in the INT4 implementation, which is *25.45*% and *33.87*% lower compared to INT8 and INT16 respectively. As for the LUT, it is *22.82*% and *38.26*% lower than INT8 and INT16 implementations. Consequently, the INT4 implementation is the one that requires less resources due to the reduced bit-precision.

Table 4 Resource usage percentage and latency for each used bit-precision of the evaluation CNN.

Resource Utilisation (%)	**INT4**	**INT8**	**IN16**
BRAM	1.13	1.51	1.89
DSP	1.0	1.0	1.0
FF	0.41	0.55	0.62
LUT	0.71	0.92	1.15
Latency (x1000 cycles)	181.9	182.1	239.8

Diving a bit further, both INT4 and INT16 implementations of the evaluation CNN are then optimized by applying loop swapping (*LS*) only, as in Section 4.3. Figure 9 compares non-optimized (no loop swapping - *NLS*) and optimized implementations of INT4. Resource usage of both implementation is about the same. However, the latency of the INT4 *LS* implementation is *2.89*% faster compared to the non-optimized INT4.

Figure 10 shows the implementation results of both *NLS* and *LS* implementations. From this figure, one can seen that the LUT, FF and the DSP usage are about the same in both implementation. The BRAM usage is *29*% larger in the *LS* implementation compared to the *NLS* one. Regarding the latency, *LS* is *2.2*% faster compared to *NLS*, this is because these loop optimizations improves memory accesses by moving data closer to the computing part.

5 Evaluation on a State-of-the-art CNN

Beyond the simple evaluation CNN, a more complex CNN was used to assess the features of the proposed hardware generation approach. For this purpose, MobileNet-V2 [26] was used as a workload. It is a modular CNN by means of the hyperparameter α. In the context of this paper, α is set to *0.25* and the input image has a dimension of $3 \times 128 \times 128$. The MobileNet-V2 CNN was trained using the Imagenet dataset [2] and was later quantized using a 8-bit precision.

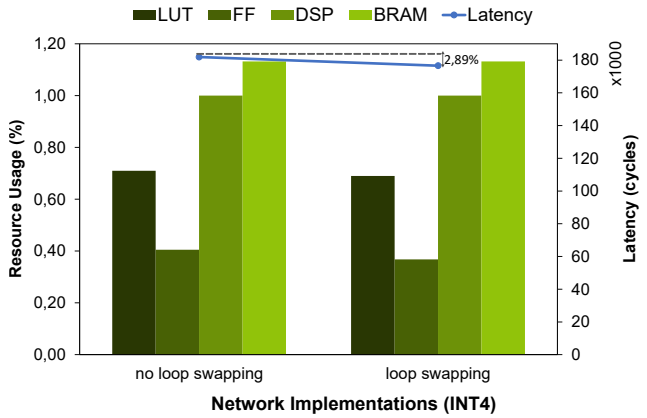

Fig. 9 Non-optimized (*NLS*) and optimized (*LS*) implementations of the evaluation CNN quantized into 4-bit.

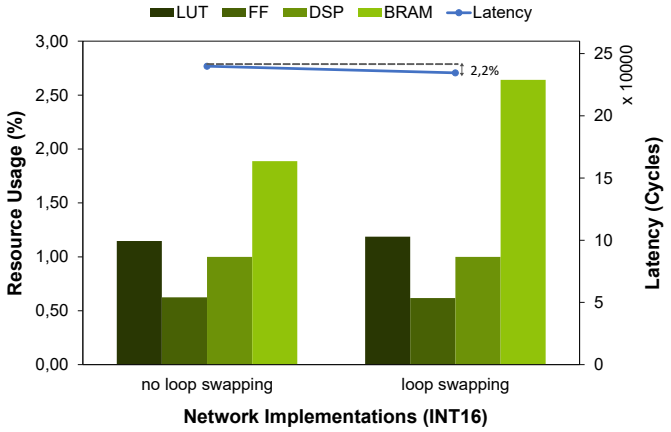

Fig. 10 Non-optimized (*NLS*) and optimized (*LS*) implementations of the evaluation CNN quantized into 16-bit.

5.1 Characterization of MobileNet-V2

The selected CNN is characterized by the first step of the proposed framework. Therefore, layer-wise and network-wise metrics are computed that will drive the hardware generation step to generate an optimized accelerator.

Layer-wise, various metrics can be computed, such as the width/depth comparison, the pixel reuse percentage and the memory requirements.

For instance, Figure 11 illustrates the evolution of the width and depth of feature maps (in number of pixels) in each layer as well as the pixel reuse (in percentage). One can see that both the width and depth progress in opposite directions. The width decreases cross the CNN due to the successively applied convolutions along the network. On the other hand, the depth grows wider due to the large number of filters applied throughout the CNN.

This metric helps determining the loops order in convolutional and pooling layers, especially when combined with the pixel reuse metric. As it can be seen in Figure 11, the pixel reuse is not consistent along the CNN, since it varies according to the employed stride value as well as the kernel size. For instance, layers having a 1×1 kernel dimension use all pixels equally, thus the reuse percentage is 100%. Layers with a 3×3 kernel size have lower reuse percentage compared to $conv1\times1$ layers. For example, a $conv1$ has a stride of 2, its reuse percentage is 23.10%. Layer $conv3\_2\_3\times3$ has a stride of 1, and thus the reuse percentage is higher (72%) compared to $conv1$. Another aspect is the depth of the feature map, which helps identifying parallelism opportunities where relevant loop optimizations could be applied. Therefore, channels parallelism could be encouraged in deeper layers where the depth exceeds the width.

Based on this analysis, layers of the MobileNet-V2 CNN are implemented using different loops order. Layers having low reuse percentage are implemented using the R, C, M, N, Kx, Ky loop order. All $conv1\times1$ are implemented by changing the loops order to M, R, C, N, Kx, Ky to encourage reuse and limit data transfers since all activations are equally used, this also promotes channels parallelism. Early layers should be implemented using the M, R, C, N, Kx, Ky order since the width of the input feature maps is important, especially in layers $conv1$ to $conv3\_2\_1\times1$. Applying this transformation on those layers encourages reuse by moving the data closer to the computational part (kernel loops $[Kx, Ky]$).

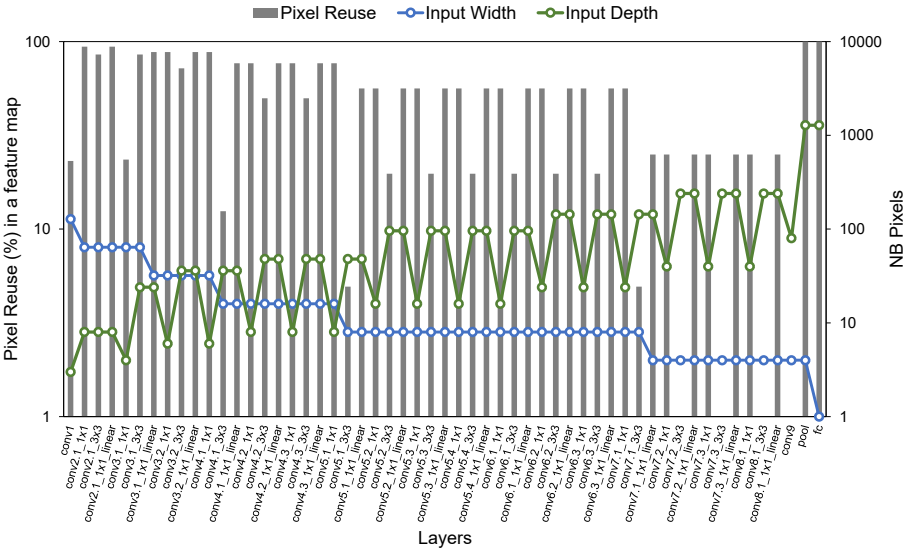

Fig. 11 Width and depth comparison as well as the pixel reuse percentage of each layer of the MobileNet-V2 CNN. Width and depth are expressed in number of pixels.

Memory requirements for both weights and input features are also computed. Figure 12 presents these memory needs in MobileNet-V2. Some of these

layers does not have any parameters like pooling and *conv_sum*. This latter is only used to sum the outputs of *conv_linear* layer and the residual block.

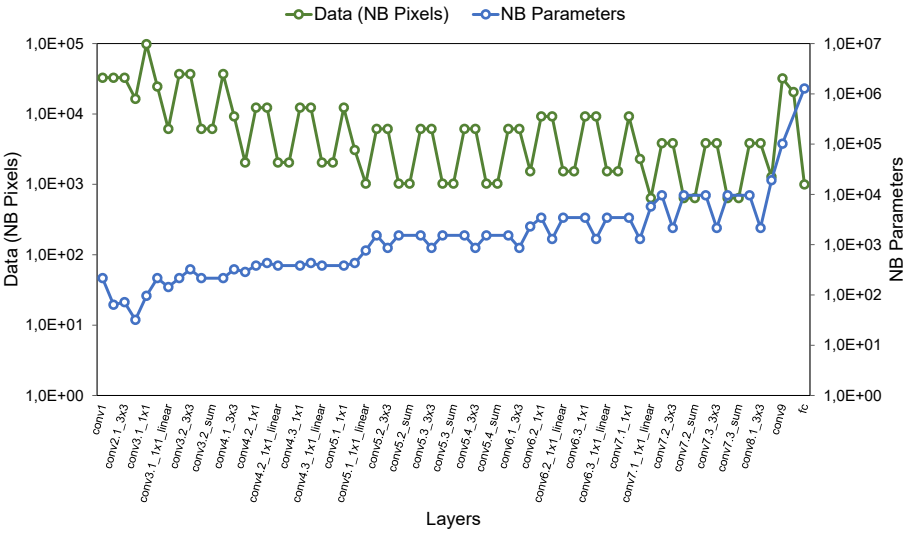

Fig. 12 Per layer memory requirements of both weights and activations in MobileNet-V2. Activations are expressed in number of pixels.

Another interesting metric is the connection graph of the CNN. It allows to find similarities between layers to later reduce the resource usage of the overall network. Figure 13 illustrates the connection graph of the MobileNet-V2 CNN. Layers are grouped into two categories of *Conv_blocks*, each one has a different stride value. Figure 14 sketches these *Conv_blocks* where the first one (on the left) is a residual block with a stride of 1, and the second one (on the right) with is stride of 2 is for downsizing. The connection graph highlights the *Conv_blocks* that have the same kernel dimensions and the same memory requirements, in terms of number of input and output pixels. It is leveraged to reduce the hardware resources of the CNN by instantiating one C function for multiple identical layers. Therefore, the *Allocation* pragma can be used to enable sharing the same RTL resources during HLS. For instance, the three *Conv1*$\times$*blocks* (i.e. three connections between the two green blocks) in the large black-box labeled 6, use the same RTL resources. The same applies for others blocks highlighted with the same color. It is worth noting that similar layers exist in different *Conv1*$\times$*blocks* from other black black-boxes. These layers are not presented and are omitted for simplicity reasons. However, memory requirements presented in Figure 12 help identifying these layers. For example, *conv4.3_1*$\times$*1* and *conv5.1_1*$\times$*1* have the same memory requirement in terms of weights and input activations.

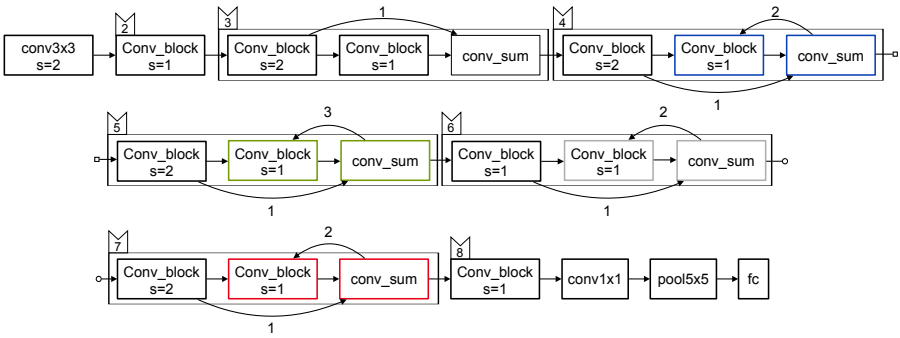

Fig. 13 Connection graph of MobileNet-V2, where convolutional blocks having the same memory requirements are highlighted using the same color.

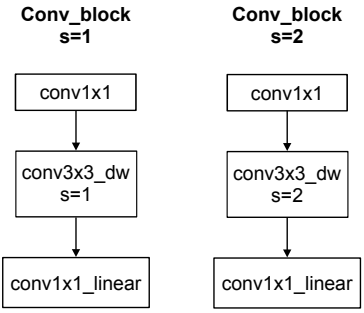

Fig. 14 Convolutional blocks in MobileNet-V2 using different stride value, where s is the *Stride*. Each block consists of three convolutions: conv1x1, depth-wise conv3x3 ($s = 1$ or $s = 2$) and one conv1x1_linear.

5.2 Hardware accelerator generation of MobileNet-V2

Using the proposed framework, the metrics obtained by the characterization phase are leveraged to guide the generation process of the optimized accelerator targeting MobileNet-V2. Various implementations are realized : per layer, and full implementations. Since MobileNet-V2 is a large network, a larger FPGA is required to fit the whole CNN. Therefore, all implementations (per layer and full implementations) are synthesized using a frequency of 100MHz targeting the Xilinx Kintex xcku035 FPGA, which is a larger FPGA than the one used to implement the evaluation CNN.

5.3 Generation of convolutional layers

Some layers are implemented separately to highlight the impact of various code transformations. Therefore, 4 different layers are selected based on their position in the network. These layers are: *conv1*, *conv2.1_3×3*, *conv8.1_1×1* and *conv9*. For each layer, different implementations are performed: *NLS* and *LS* are performed for all layers, and *LSP* for *3×3* convolutional layers. In the *LSP* instance, kernel loops are unrolled due to their small bounds

($1 < loop\ bound \leq 3$). It is worth noting that the LS implementation means that the loop order is changed from R, C, M, N, Kx, Ky to M, R, C, N, Kx, Ky, and is applied according to the characterization results explained in Section 5.1.

Implementations results are presented in Figure 15. The Loop swapping (LS) in the presented layers shows a slight improvement in performance. For instance, in the $conv1$ the LS is 1.5% faster than the NLS implementation. In addition, it uses 1.22% less LUT and 5.45% less FF compared to the NLS implementation. The BRAM and DSP utilization is the same for both NLS and LS. The LS implementation of the $conv2.1\_3\times3$ layer is 0.57% faster compared to the NLS and consumes 0.57% less FF. However, the LUT resource utilization increased by 5.32%. Regarding $conv8.1\_1\times1$ and $conv9$ layers, the performances in the LS implementation are respectively 0.39% and 0.19% faster compared to the NLS implementation. As for the resource utilization in the LS, $conv8.1\_1\times1$ uses 12.32% less LUT and 12.87% more FF. The same goes for $conv9$ which uses 1.36% less LUT and 13.2% more FF. The performance improvement is the result of the chosen loops order that allowed to limit unnecessary data transfers and enhanced data locality, which explains the increase in the number of used FF.

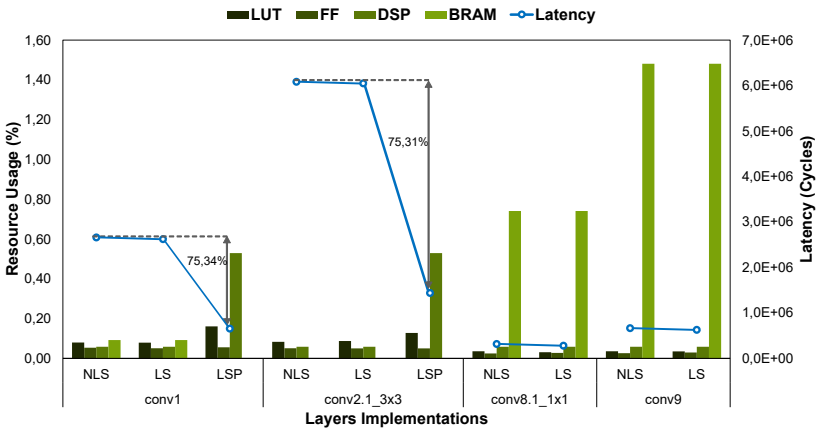

Fig. 15 Results of each of the selected layers of MobileNet-V2.

The LSP optimization is 75.34% faster for $conv1$, and in 75.31% faster for $conv2.1\_3\times3$ compared to NLS. Regarding resource utilization, LSP consumes more resources than NLS in both optimized layers implementations, especially in terms of DSPs and LUTs. For example, LSP consumes 9× more DSPs due to the unrolling of kernel loops in $conv1$ and $conv2.1\_3\times3$ which requires more DSPs. In addition, LSP uses 2× more LUTs compared to NLS in $conv1$, and 1.5× more LUTs in $conv2.1\_3\times3$. The global increase in resources can be explained by the fact that unrolling loops creates an RTL copy of the loop body for each iteration to simultaneously run the whole loop.

5.4 Full accelerator hardware generation

Different implementations were realized to implement the entire MobileNet-V2 CNN: NLS (as a reference implementation), LS, LS with allocation (referred as LSA), LSP and LSP with allocation (referred as $LSPA$). The allocation pragma, explained in 4.2.3, allows using the same RTL resources for multiple similar layers.

In Figure 16, LS is 0.12% faster than NLS and uses on average 0.019% more resources. Applying the allocation pragma with LS (LSA) improves performance by 3.9% and leads to 0.69% less resource usage compared to LS. LSP is 74.05% faster than NLS, which comes at a cost of using, on average, 3.26% more resources. The same applies for $LSPA$ which improves performance by 74.51% and consumes 2.62% more resources compared to a standard implementation. However, $LSPA$ uses 0.26% less resources than LSP thanks to the allocation pragma.

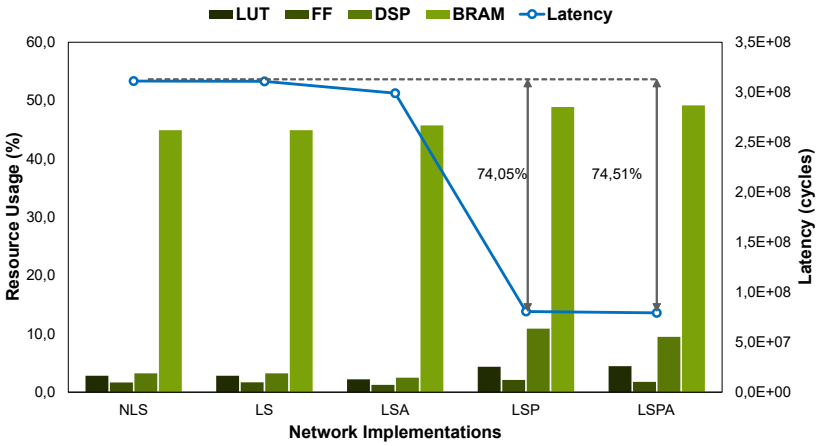

Fig. 16 Results of each implementation of MobileNet-V2.

Table 5 summarizes the results of the various implementations of MobileNet-V2. As it can be seen from Figure 16 and Table 5, improving performance comes with an area cost. On large CNNs, area cost can be reduced by using the same RTL instance to implement similar layers.

Regarding the needed time to get the targeted solution, it depends on the size of the input CNN, since the hardware synthesis is time-consuming. In general, it would take about 2 to 4 hours.

6 Conclusion

This paper introduced a hardware generation framework enabling to design efficient FPGA accelerators for CNNs. It extracts relevant knowledge about the CNN computation behavior in the form of hardware-aware metrics to drive a

Table 5 Resource usage (in percentage) and latency (in cycles) for different implementations of MobileNet-V2.

Resource Usage (%)	NLS	LS	LSA	LSP	LSPA
LUT	2,816	2,811	2,193	4,359	4,453
FF	1,665	1,700	1,247	2,103	1,763
DSP	3,235	3,235	2,471	10,882	9,471
BRAM	44,907	44,907	45,741	48,889	49,167
Latency (cycles)	3,11E+08	3,11E+08	2,99E+08	8,07E+07	7,93E+07

hardware generation phase by exploiting CNN algorithm specifications. Using these results, the framework applies source code optimizations (e.g. loop swapping) and selects their relative right parameters. Other targeted optimizations use tool-specific pragmas to generate more efficient accelerators. Using an evaluation CNN, the framework automatically generates an accelerator which is 13% faster than using relevant optimized pragmas only. It is worth noting that performing such manual optimizations is already a time-consuming task. On a state-of-the-art network, the framework generated an accelerator 74.51% faster than applying *LS* only, while using a reasonable amount of resources.

Future works will focus on generating accelerators for other state-of-the-art CNNs having large and complex topologies. Efforts will also be put on exploring the space of new tool-specific pragmas to automatically choose the right ones with respect to the design goals. Moreover, targeting sparse CNNs by establishing HLS-based compression techniques will also be pursued.

References

[1] Lecun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. In: Proceedings of the IEEE, pp. 2278–2324 (1998)

[2] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L.: Imagenet large scale visual recognition challenge. International Journal of Computer Vision (IJCV) **115**(3), 211–252 (2015). https://doi.org/10.1007/s11263-015-0816-y

[3] Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: Efficient convolutional neural networks for mobile vision applications. CoRR **abs/1704.04861** (2017) https://arxiv.org/abs/1704.04861

[4] Ali, N., Philippe, J.-M., Tain, B., Peyret, T., Coussy, P.: Deep neural networks characterization framework for efficient implementation on embedded systems. In: 2020 IEEE Workshop on Signal Processing Systems (SiPS), pp. 1–6 (2020). https://doi.org/10.1109/SiPS50750.2020.9195227

[5] Lahti, S., Sjövall, P., Vanne, J., Hämäläinen, T.D.: Are we there yet?

a study on the state of high-level synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **38**(5), 898–911 (2019). https://doi.org/10.1109/TCAD.2018.2834439

[6] Ali, N., Philippe, J.-M., Tain, B., Coussy, P.: Exploration and generation of efficient fpga-based deep neural network accelerators. In: 2021 IEEE Workshop on Signal Processing Systems (SiPS), pp. 123–128 (2021). https://doi.org/10.1109/SiPS52927.2021.00030

[7] Whatmough, P.N., Zhou, C., Hansen, P., Venkataramanaiah, S.K., Seo, J., Mattina, M.: Fixynn: Efficient hardware for mobile computer vision via transfer learning. In: The 2nd Conference on Systems and Machine Learning (SysML) (2019). https://arxiv.org/pdf/1902.11128

[8] Carbon, A., Philippe, J.-M., Bichler, O., Schmit, R., Tain, B., Briand, D., Ventroux, N., Paindavoine, M., Brousse, O.: PNeuro: A scalable energy-efficient programmable hardware accelerator for neural networks. In: 2018 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1039–1044 (2018). https://doi.org/10.23919/DATE.2018.8342165

[9] G. Desoli et al.: 14.1 a 2.9tops/w deep convolutional neural network soc in fd-soi 28nm for intelligent embedded systems. In: 2017 IEEE International Solid-State Circuits Conference (ISSCC), pp. 238–239 (2017). https://doi.org/10.1109/ISSCC.2017.7870349

[10] Han, S., Mao, H., Dally, W.J.: Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. International Conference on Learning Representations (ICLR) (2016)

[11] Chen, Y.-H., Krishna, T., Emer, J.S., Sze, V.: Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE Journal of Solid-State Circuits **52**(1), 127–138 (2017). https://doi.org/10.1109/JSSC.2016.2616357

[12] Wang, C., Gong, L., Yu, Q., Li, X., Xie, Y., Zhou, X.: Dlau: A scalable deep learning accelerator unit on fpga. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **36**(3), 513–517 (2017). https://doi.org/10.1109/TCAD.2016.2587683

[13] Lu, L., Xie, J., Huang, R., Zhang, J., Lin, W., Liang, Y.: An efficient hardware accelerator for sparse convolutional neural networks on fpgas. In: 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 17–25 (2019). https://doi.org/10.1109/FCCM.2019.00013

[14] A. Solazzo et. al: Hardware design automation of convolutional neural networks. In: 2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp. 224–229 (2016). https://doi.org/10.1109/ISVLSI.2016.101

[15] Shen, Y., Ferdman, M., Milder, P.: Maximizing cnn accelerator efficiency through resource partitioning. In: 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), pp. 535–547 (2017). https://doi.org/10.1145/3079856.3080221

[16] Rivera-Acosta, M., Ortega-Cisneros, S., Rivera, J.: Automatic tool for fast generation of custom convolutional neural networks accelerators for

fpga. Electronics **8**(6) (2019)

[17] Y. Guan et. al: FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In: 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 152–159 (2017). https://doi.org/10.1109/FCCM.2017.25

[18] Umuroglu, Y., Fraser, N.J., Gambardella, G., Blott, M., Leong, P., Jahre, M., Vissers, K.: Finn: A framework for fast, scalable binarized neural network inference. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. FPGA '17, pp. 65–74 (2017). https://doi.org/10.1145/3020078.3021744

[19] F. Fahim et al.: hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices. CoRR **abs/2103.05579** (2021) https://arxiv.org/abs/2103.05579

[20] Erdem, A., Silvano, C., Boesch, T., Ornstein, A., Singh, S., Desoli, G.: Design space exploration for orlando ultra low-power convolutional neural network soc. In: 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 1–7 (2018). https://doi.org/10.1109/ASAP.2018.8445096

[21] R. Venkatesan et al.: Magnet: A modular accelerator generator for neural networks. In: 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 1–8 (2019). https://doi.org/10.1109/ICCAD45719.2019.8942127

[22] List, C.: N2D2 - Neural Network Design & Deployment. Manual available on Github. https://github.com/CEA-LIST/N2D2/

[23] Fei-Fei, L., Fergus, R., Perona, P.: Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. In: 2004 Conference on Computer Vision and Pattern Recognition Workshop, pp. 178–178 (2004). https://doi.org/10.1109/CVPR.2004.383

[24] 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 '15, pp. 161–170. Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2684746.2689060. https://doi.org/10.1145/2684746.2689060

[25] Rahman, A., Oh, S., Lee, J., Choi, K.: Design space exploration of fpga accelerators for convolutional neural networks. In: Proceedings of the Conference on Design, Automation & Test in Europe. DATE '17, pp. 1147–1152. European Design and Automation Association, Leuven, BEL (2017)

[26] Sandler, M., G. Howard, A., Zhu, M., Zhmoginov, A., Chen, L.: Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. CoRR **abs/1801.04381** (2018) https://arxiv.org/abs/1801.04381