Exploration and Generation of Efficient FPGA-based Deep Neural Network Accelerators

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

Exploration and Generation of Efficient FPGA-based Deep Neural Network Accelerators

Nermine Ali, Jean-Marc Philippe, Benoit Tain
Université Paris-Saclay, CEA, List
F-91120, Palaiseau, France
firstname.lastname@cea.fr

Philippe Coussy
University of South Brittany
Lorient, France
philippe.coussy@univ-ubs.fr

***Abstract*—Convolutional Neural Networks (CNNs) have emerged as an answer to next-generation applications such as complex image recognition and object detection. Embedding such compute-intensive and memory-hungry algorithms on edge systems will lead to smarter high-value applications. However, the algorithmic innovations in the CNN field leave the hardware accelerators one step behind. Reconfigurable hardware (e.g. FPGAs) allows designing custom accelerators adapted to new algorithms. Furthermore, new design approaches such as high-level synthesis (HLS) enable to generate RTL code based on high-level function descriptions. This paper presents a high-level CNN accelerator generation framework for FPGAs. A first phase of the framework characterizes CNN descriptions using hardware-aware metrics. These metrics then drive a hardware generation phase which builds the proper C source code implementation for each layer of the network. Finally, an HLS tool outputs the synthesizable RTL code of the accelerator. This approach aims at reducing the gap between the evolving applications based on artificial intelligence and hardware accelerators, thus reducing time-to-market of new systems.**

***Index Terms*—Convolutional Neural Networks, Design Space Exploration, High Level Synthesis, Embedded Systems, FPGA**

I. Introduction

Deep learning algorithms, such as Convolutional Neural Networks (CNNs) [1], are promising tools to tackle the challenges of next-generation applications such as complex image recognitions, classifications and object detections. These Deep Neural Networks (DNNs) are composed of formal neurons organized in several computational layers. The first layers are mainly based on convolutional filters that act as feature extractors and the last ones are based on fully connected layers that perform a classification process thanks to the extracted features. The parameters of the different layers (filter coefficients, weights, etc.) are learned during a preliminary supervised training phase on specific data.

The different layers perform mathematical operations such as convolution filters, pooling, weighted sum and activation functions (Fig. 1). Convolution layers extract features by applying weighted filters on input feature maps. Non-linear mathematical functions (such as the rectified linear unit or ReLU), called activation functions, define the way a neuron is activated (i.e. the input value has an action on the prediction) or not. To decrease the number of parameters, pooling operations are inserted to reduce the size of the feature map by maintaining robust features only. The final layers, usually based on fully connected (FC) layers, classify the input data.

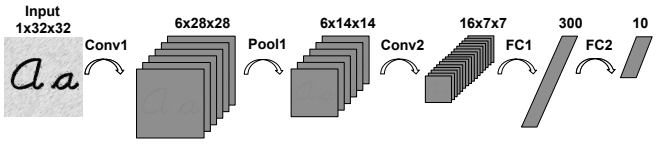

Fig. 1. Typical CNN structure (from [1]).

These algorithms compete on popular image recognition challenges such as the ImageNet Large Scale Visual Recognition Challenge [2]. New topologies and layers are continuously published to either improve the top-1 and top-5 scores or to reduce the computation and memory requirements. Optimization techniques such as pruning or quantization are also proposed to augment DNN sparsity or reduce the precision of data and parameters to target edge systems. Hardware accelerators are designed to support these new techniques, with tradeoffs between efficiency and flexibility. Thus, the gap between the algorithmic optimizations and designed hardware architectures is still there. In [3], the idea of characterizing the algorithm to help hardware designers with relevant target-agnostic but hardware-aware metrics was introduced. These metrics derive analysis and hints on mapping strategies and/or configurations of an accelerator. Beyond the configuration of a DNN accelerator template, one can also use these metrics to optimize the generation of an accelerator using new design approaches such as high-level synthesis (HLS).

This paper proposes a two-phase approach for generating efficient FPGA-based DNN accelerators. The first phase characterizes the DNN behavior and extracts its relevant hardware-related features while the second phase is a hardware accelerator generator module based on HLS. The idea is to reduce the gap between software and hardware by lowering algorithmic descriptions using hardware-aware metrics and abstracting hardware based on high-level hardware descriptions. The paper is organized as follows. Section II discusses related work. Section III presents the overall hardware accelerator generation framework including the characterization part. Section IV presents the evaluation of the framework on a full CNN while section V concludes the paper and introduces future works.

II. Related Work

A wide range of hardware accelerators for deep learning applications has been developed with the objective to reach both high performance and energy efficiency. Different approaches are proposed with flexibility as a tradeoff, from deeply optimized fixed structures for feature extraction [4] to programmable DSP-like architectures, either homogeneous [5] or heterogeneous [6]. Some proposals also exploit features resulting from either the DNN algorithm itself (sparsity, etc.) or specific hardware-aware optimization techniques to improve the energy efficiency of the overall system. One can cite pruning [7] or compression techniques [8].

Reconfigurable architectures such as FPGAs are also interesting targets for deep learning accelerators because their internal structure is suited to both the dataflow nature of DNN algorithms and their intrinsic spatial parallelism. Fine-grain reconfigurability property of FPGAs enables to change both the design of the accelerator depending on the topology of the DNN and to size the computing resources and memory to the needed precision. Both FPGA vendors and academic researchers propose different approaches and tools to perform deep learning. [9] introduces a pipeline architecture with tiling techniques to improve computation and [10] investigates the use of FPGAs to design an accelerator for sparse CNNs.

To tackle the challenges related to the quickly changing landscape of deep learning algorithms, it is therefore paramount to work on new design methodologies for DNN accelerators, using hardware abstraction and/or design space exploration. Some works propose direct hardware generation frameworks based on a network description or configuration. They generate a synthesizable high-level code using commercial HLS tools such as Catapult or Vivado-HLS. For example, the web-based framework presented in [11], targeting FPGAs, provides an empirical estimation of the hardware resources utilization based on a network configuration. The work in [12] proposes a framework that generates an accelerator for a custom CNN configuration. FP-DNN, based on a model description, instantiates RTL-HLS hybrid templates to generate CNN accelerators using HLS-tools [13]. FINN targets binarized NN and generates an optimized C++ description of a streaming architecture based on a frame-per-second target and a trained BNN model [14]. Other approaches use design space exploration of existing architectures [15] or architectural templates [16]. Unfortunately, these approaches do not fully leverage the potentials of HLS in designing a tailored hardware architecture based on high-level algorithmic descriptions.

This work offers another path between straightforward architecture generation and time-consuming exploration by proposing to use hardware-aware application characterization as a first step to provide hints to the high-level code generation phase. The goal is to provide an automatic HLS-based tool-independent hardware generation framework that can output an optimized high-level representation of an accelerator based on a characterization of the input neural network. Next section explains this approach and the proposed flow.

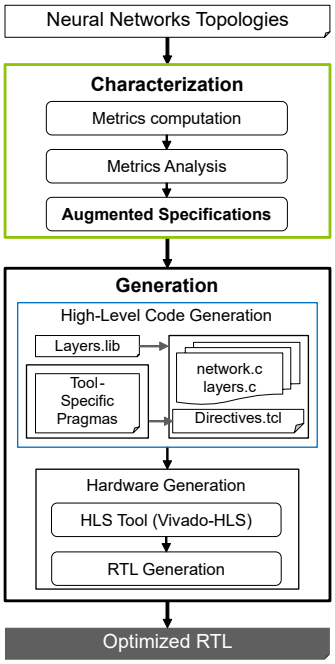

Fig. 2. Overview of the proposed exploration and accelerator generation framework flow.

III. Overview of the proposed DNN Exploration and Accelerator Generation Framework

Previous work [3] introduced a characterization framework that aims at capturing the behavior of DNN algorithms to help configuring an accelerator architecture and/or choosing between mapping strategies. The present work proposes to use this characterization phase to quickly generate an optimized RTL code from high-level function descriptions by exploiting the gathered metrics. With the hardware generation being driven by the extracted metrics, finding a viable link between the application specifications and the hardware implementation is a very challenging problem.

A. Presentation of the approach

Fig. 2 presents the high-level view of the proposed approach. The flow is composed of two main modules: the characterization framework introduced in [3] followed by a hardware accelerator generation module based on HLS. The input of the flow is a description file of the CNN (i.e. its topology and the types and hyperparameters of the different layers). This description file can also embed the parameters of the CNN (filter coefficients, weights), depending on the analysis to be performed. Typically, description files from classical deep learning frameworks such as TensorflowLite or de facto standards such as ONNX can be used.

The first step is the network characterization phase which extracts relevant metrics from neural network descriptions. Then, the tool analyzes these metrics and combines them to derive assumptions on implementation strategies and configuration of the targeted hardware architecture to execute the applications (e.g. mapping algorithms to optimize data

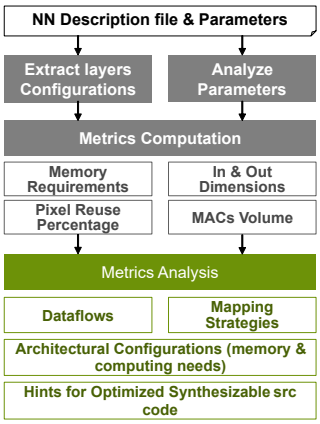

Fig. 3. Overview of the characterization phase.

movement). The second step is the generation phase. It exploits the extracted metrics and the analysis to generate an optimized HLS-ready C code which will be transformed into an optimized RTL code after being fed to an HLS tool. More details on these phases are given in the following subsections.

B. Characterization and analysis phase

The characterization phase is depicted in Fig. 3. The tool extracts the configurations of the layers and can optionally analyze the CNN parameters. It then computes relevant metrics that characterize the CNN on a layer-by-layer basis, such as the memory requirements, the dimensions of the feature maps, the percentage of pixel reuse or the volume of MAC operations. Other metrics related to the parameters can be computed to derive the sparsity of the DNN, the distribution of the weights, their precision, etc. This phase also generates charts and graphs to provide the user with synthetic data.

These measurements are then passed to an analysis module to derive conclusions or at least hints related to interesting options to include in the mapping strategies or techniques to improve the architecture. For example, the analysis of the sparsity of the DNN can lead to use compression techniques to either reduce memory requirements or to speed up the computations. The pixel reuse percentage helps to choose the right dataflow to optimize energy efficiency and memory hierarchy, and the dimensions of the input and output feature maps allow to apply optimized tiling strategies.

In this work, the results of this phase are used to both drive the code generation for HLS-based implementations and to choose the right pragmas for an HLS tool. For example, the reuse results combined with the widths and depths of the input feature maps are used to determine which loop levels to tile and the tiling parameters of the convolution implementation.

C. Accelerator generation phase

This phase generates an optimized C code of the network using a library of layers. The computed metrics obtained in the first phase drive the generation of each layer by determining whether or not to tile certain loops, in a loop nest, and the appropriate tiling parameters to employ. Loops order in a layer can also be determined. In addition, these metrics also dictate tool-specific pragmas to optimize the overall performance of the resulting CNN accelerator and to have a tradeoff between latency and resources utilization. Finally, an HLS tool generates the optimized RTL representation of the CNN, ready for synthesis, from the C source files.

IV. Framework evaluation

This section presents the evaluation of the presented approach to obtain an optimized accelerator for a given DNN. The generic flow from Fig. 2 was instantiated using N2D2 DNN description files and Vivado-HLS, an HLS tool targeting Xilinx FPGAs. N2D2 (Neural Network Design and Deployment) [17] is an open source framework for building DNN-based applications, from learning to code generation for several targets. It also includes specific features targeting embedded systems such as precision reduction techniques.

A small CNN, presented in Fig. 4, was implemented using the N2D2 framework for evaluation purposes. The six layers are: 3x3 and 5x5 2D-convolution, 3x3 and 5x5 pooling and 2 fully-connected layers. This evaluation CNN was trained with N2D2 on the Caltech-101 dataset [18] to classify four categories of images (airplanes, car sides, faces, motorbikes). The obtained accuracy is 95.31% on the validation database. 8-bit integer format was used for both pixels and parameters during the training phase. Thus, $int8$ data format was used in the C-HLS source code for the hardware generation.

A. Characterization of the evaluation CNN

The characterization phase delivers a per-layer analysis of the computational complexity and the memory requirements, as shown in Table I. Data-related metrics are also computed: the maximum data reuse for each convolutional layer and a width/depth comparison of input feature maps for all layers. Fig. 5 shows these results and illustrates the decrease in the frequency of pixel usage along the CNN due to the reduction of the dimensions of the feature maps in deeper layers. It also shows the evolutions of the widths and depths of the input feature maps throughout the CNN using the number of pixels as a measurement unit.

TABLE I
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
$conv1\_3\times3$	41472	18	4608
$pool1\_3\times3$	4608	0	512
$conv1\_5\times5$	14400	100	288
$pool1\_5\times5$	288	0	32
$fc1$	2048	2048	64
$fc2$	256	256	4
$softmax$	4	0	4
Total	58176	2422	5512

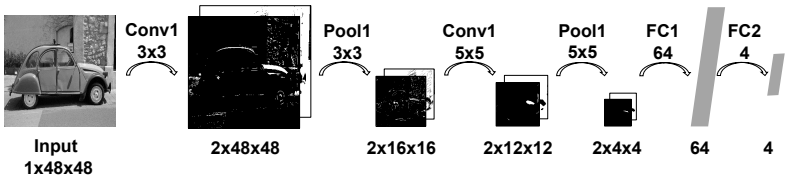

Fig. 4. Architecture of the evaluation CNN.

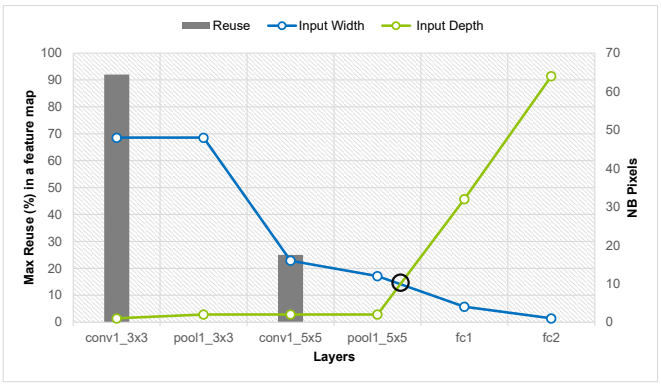

Fig. 5. Widths and depths comparison of the input feature maps in each layer of the evaluation CNN together with pixel reuse percentage.

B. Hardware accelerator generation for the evaluation CNN

The computed metrics are leveraged to drive the generation of an optimized accelerator for the evaluation CNN. Each layer is implemented separately to test different configurations based on the position of the layer in the CNN. All implementations were performed at a frequency of 100MHz using the Xilinx Zynq7000 xc7z030 FPGA as a target.

1) Generation of convolutional layers: A commonly used approach to implement a convolutional layer is presented in Listing 1. Based on the generated metrics, this implementation can be enhanced by applying two different transformations: loop tiling and loop reordering.

```
ifmap[N][(R-1)*S+K][(C-1)*S+K] //input maps
outfmap[M][R][C]   //output maps
weights[M][N][K][K]
10:for(r=0; r<R; r++)     //output X
 11:for(c=0; c<C; c++)    //output Y
  12:for(m=0; m<M; m++)       //nb outputs
   13:for(n=0; n<N; n++)      //nb channels
    14:for(kx=0; kx<Kx; kx++)     //kernel X
     15: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 1. Pseudo-code of a convolutional layer.

2) Loop tiling and ordering: Loop tiling promotes data locality and improves performance by reducing unnecessary data accesses and transfers if the tile size is correctly set. Loop tiling introduces new loops as shown in Listing 2 with different tiling parameters T. However, determining which loop level to tile and its tiling parameter is challenging since it is based on different input and output dimensions. One can use the results of the characterization process to generate an optimized source code that best fits each layer. The related metrics are the reuse percentage and the width/depth evolution comparison. When combined, they can highlight the loop levels to be tiled. In addition, they set thresholds for the tiling parameters in order to limit the design space as in (1). The minimum threshold values for the height C and the width R are set to the kernel height Ky and width Kx respectively.

$$\begin{aligned} Kx \leq Tr \leq R \\ Ky \leq Tc \leq C \\ tn \leq Tn \leq N \\ tm \leq Tm \leq M \end{aligned} \tag{1}$$

For example, the first layer, *conv1_3×3*, of the CNN in Fig. 4 has the biggest input and output feature maps. According to Fig. 5 it also has the highest reuse percentage (92%), the highest width (48) and the lowest number of channels (1). Analyzing these metrics, two actions could be taken: choosing the loop level to tile and determining the tiling parameter. Here, loops *l0* and *l1* are good candidates to tile to improve data locality and reuse of the input feature map by allowing a portion of the image to be stored on-chip. The resulting code is presented in Listing 2.

```
ifmap[N][(R-1)*S+K][(C-1)*S+K] //input maps
outfmap[M][R][C]   //output maps
weights[M][N][K][K]
10:for(r=0; r<R; r+=Tr)     //output X
 11:for(c=0; c<C; c+=Tc)    //output Y
  12:for(m=0; m<M; m++)       //nb outputs
   13:for(n=0; n<N; n++)      //nb channels
    10.1:for(tr=r; r<min(R,r+Tr); tr++)
     11.1:for(tc=c; c<min(C,c+Tc); tc++)
      14:for(kx=0; kx<Kx; kx++)     //kernel X
       15: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 2. Pseudo-code of a tiled convolutional layer.

The same approach is applied to layers *pool1_3×3* and *conv1_5×5*. Loop levels *l0* and *l1* are the ones to be tiled since the width is bigger than the depth is these layers. Different tiling parameters were tested to show the impact on performance and area. Each of these layers is implemented separately (place and route), omitting infrastructure like memory controllers and crossbars. Fig. 6 proves that raising the tiling factor improves the latency while having low impact on resource usage. Loop levels *l2* and *l3* are not tiled since the depth is very small. The reuse percentage in layer *conv1_5×5*

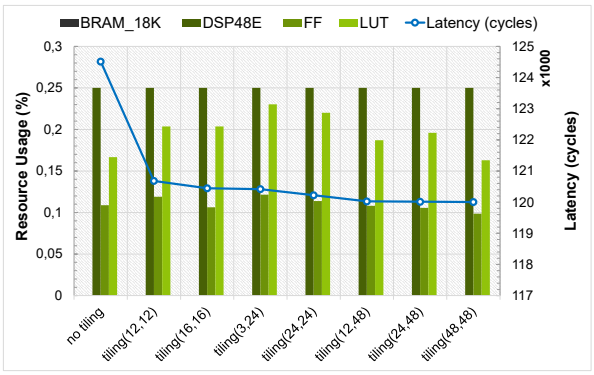

Fig. 6. Results of $conv1\_3\times3$ implemented using different tiling values.

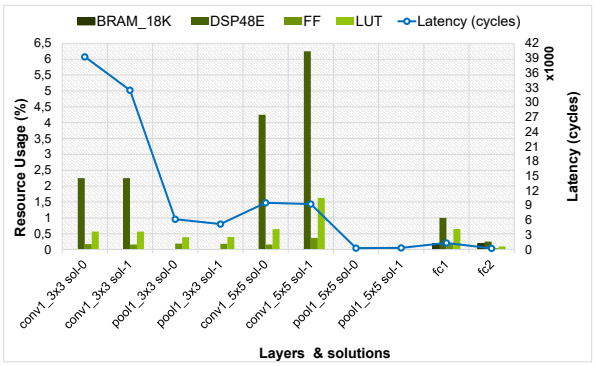

Fig. 7. Results of each optimized layer of the evaluation CNN.

is 25% and 0% in the remaining layers. This explains the low impact of tiling in these layers as it can be seen in Table II for $conv1\_5\times5$ and Table III for $pool1\_5\times5$.

Loop ordering is another data-related optimization technique which allows a certain amount of data to remain available on-chip for a period of time. Determining how to reorder the loops to improve latency follows the same rule as identifying which loop level to tile. The same tiled loops could be swapped with another loops (on a higher or a lower level) to improve performance. Choosing the order of loops relies on the input and output dimensions of the layer as well as the pixel-reuse. As an illustration, swapping ($l0$, $l1$) with $l2$ in layer $conv1\_3\times3$ has the same impact as using (48, 48) as tiling parameters which are the maximum possible values to apply. Performance and resources usage are the same. This is also valid for the other convolutional and pooling layers.

TABLE II
RESOURCE USAGE PERCENTAGE AND LATENCY FOR $conv1\_5\times5$ WITH DIFFERENT TILING PARAMETERS.

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 III
RESOURCE USAGE PERCENTAGE AND LATENCY FOR $pool1\_3\times3$ WITH DIFFERENT TILING PARAMETERS.

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

3) Applying pragmas: In HLS, tool-specific pragmas are generally used to apply transformations for RTL generation. The main performance-related optimizations to look at are *loop unrolling* and *pipelining*. Loop unrolling creates N copies of the target loop (where N is the unrolling factor) and thus allows a parallel execution of N iterations of the loop. Loop pipelining allows simultaneous execution of operations inside a loop or a function, hence reducing the initiation interval.

Boosting the performance of each layer consists in optimizing the loop nest. Since the loop swapping discussed earlier led to better performance than loop tiling, the pragmas will be applied on the layers having ($l0$, $l1$) loops swapped with $l2$ loop to push the performance forward. For short, let us name this optimization LSP, for Loop Swapping and Pragmas. Applying pragmas to drive the generation phase takes into consideration the topology of the CNN. For example, the framework unrolls loop levels having a very small number of iterations, typically layers having few channels and considerably small kernel sizes. For this reason, kernel loops ($l4$, $l5$) are completely unrolled in the first four layers of the network as well as output loops ($l0$). This is not applicable in fully-connected layers where the number of outputs in $fc1$ (input of $fc2$) is very high (equal to 64). This would results in a significantly large area due to the massive use of DSPs. In this case, the innermost-loops are pipelined to maintain an acceptable performance and area. The same pragmas were applied on the same loop levels in the non-optimized (i.e. no loop transformation) layers, for short $NLSP$, where there is no loop swapping and only pragmas are applied. The results of the implementations (i.e. Vivado-HLS *solutions*) are presented in Fig. 7. Loop swapping/tiling showed no benefits in layers $fc1$ and $fc2$, for this reason only pragmas were applied.

The LSP optimization results in 0.34% less global resources usage and is about 17.34% faster than $NLSP$ optimization for $conv1\_3\times3$. This is not the case for $pool1\_3\times3$ where LSP optimizaton is faster, but consumes 0.33% more resources. The same behaviour is observed in $conv1\_5\times5$ where LSP is 2.76% faster and uses 38.7% more resources. Regarding $pool1\_5\times5$, $NLSP$ is 12% faster and consumes 50% more resources than LSP. The higher performance obtained using $NLSP$ can be explained by the small dimensions

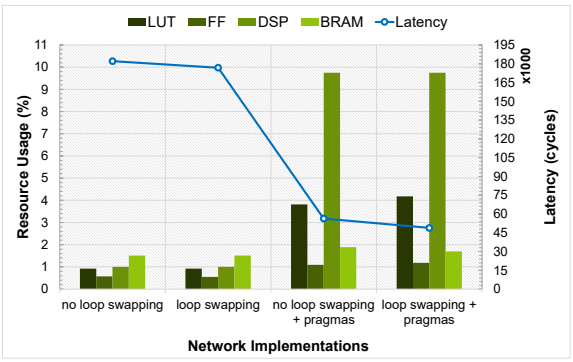

Fig. 8. Results of each implementation of the evaluation CNN.

of the input/output feature maps. Data locality cannot be leveraged due to the low data reuse ratio in these layers.

C. Full accelerator hardware generation

Using the framework, the whole structure of the accelerator is then generated using the previously optimized layers. Different implementations were performed for evaluating the benefits of the applied strategies. The first one does not include any optimization, neither loop swapping nor pragmas (NLS). The second one uses loop swapping only (LS). The third one is implemented using pragmas only ($NLSP$) and the last one with both loop swapping and pragmas (LSP). Fig. 8 shows the results of these implementations.

In Fig. 8, LS is 3% faster than NLS and uses 0.68% less resources. Applying tool-specific optimizations for NLS and LS leads to 69% and 72% speed-up respectively which comes at a cost of using 76% more resources in both optimized implementations. Comparing the last two implementations, $NLSP$ and LSP, there is 13% speed-up in LSP and 1.55% more resource usage. $NLSP$ and LSP are 69% and 73% faster respectively compared to a standard implementation. Higher performance always comes with an area cost, thus the final implementation choice depends on the design goal.

V. Conclusion

This paper presents an FPGA-based accelerator generation framework for CNNs. It exploits DNN algorithm specifications and hardware-aware metrics to drive a hardware generation phase. Using this characterization phase, the framework is able to apply code optimizations such as loop swapping together with their right parameters as well as to use tool-specific pragmas to improve accelerator generation. On a small evaluation CNN, the framework outputs an accelerator 13% faster than using optimized pragmas only, which is already a time-consuming optimization if done manually. Future works will focus on generating accelerators for state-of-the-art CNNs having large and complex topologies. Deeper layers are meant to offer more flexibility in loop-related optimizations due to their relatively large dimensions. The use of the same RTL resources for multiple layers will be explored to fit large DNNs on the FPGA target. Efforts will also be put on exploring the space of tool-specific pragmas to automatically choose the right ones with respect to the design goals.

References

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, 1998, pp. 2278–2324.

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[3] N. Ali, J.-M. Philippe, B. Tain, T. Peyret, and P. Coussy, "Deep neural networks characterization framework for efficient implementation on embedded systems," in *2020 IEEE Workshop on Signal Processing Systems (SiPS)*, 2020, pp. 1–6.

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

[5] A. Carbon, J.-M. Philippe, O. Bichler, R. Schmit, B. Tain, D. Briand, N. Ventroux, M. Paindavoine, and O. Brousse, "PNeuro: A scalable energy-efficient programmable hardware accelerator for neural networks," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 1039–1044.

[6] 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)*, Feb 2017, pp. 238–239.

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

[8] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An and energy-efficient reconfigurable and accelerator for deep and convolutional," *IEEE J. Solid-State Circuits*, vol. VOL. 52, no. NO. 1, JANUARY 2017.

[9] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "Dlau: A scalable deep learning accelerator unit on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 513–517, 2017.

[10] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang, "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)*, 2019, pp. 17–25.

[11] A. Solazzo, E. D. Sozzo, I. De Rose, M. D. Silvestri, G. C. Durelli, and M. D. Santambrogio, "Hardware design automation of convolutional neural networks," in *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, July 2016, pp. 224–229.

[12] M. Rivera-Acosta, S. Ortega-Cisneros, and J. Rivera, "Automatic tool for fast generation of custom convolutional neural networks accelerators for fpga," *Electronics*, vol. 8, no. 6, 2019.

[13] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "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)*, 2017, pp. 152–159.

[14] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, 2017, p. 65–74.

[15] A. Erdem, C. Silvano, T. Boesch, A. Ornstein, S. Singh, and G. Desoli, "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)*, July 2018, pp. 1–7.

[16] R. Venkatesan et al., "Magnet: A modular accelerator generator for neural networks," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.

[17] CEA List. (2021) N2D2 - Neural Network Design & Deployment. Manual available on Github. [Online]. Available: https://github.com/CEA-LIST/N2D2/

[18] L. Fei-Fei, R. Fergus, and P. Perona, "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*, 2004, pp. 178–178.