Efficient HLS Implementation for Convolutional Neural Networks Accelerator on an SoC

Muhammad Sarg*, Ahmed H. Khalil*, and Hassan Mostafa*†
*Electronics and Communications Engineering Department, Cairo University, Giza 12613, Egypt
†University of Science and technology, Nanotechnology and Nanoelectronics Program, Zewail City of Science and Technology, Giza 12578, Egypt
Email: muhammad.sarg@gmail.com, ahmed.hussien60@gmail.com, and hmostafa@uwaterloo.ca

2021 International Conference on Microelectronics (ICM) | 978-1-6654-0839-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICM52667.2021.9664920

***Abstract*—Convolutional Neural Networks (CNNs) have achieved high accuracy in many applications such as image recognition and classification. However, due to their large amount of parameters and intensive required operations, general purpose processors cannot achieve the desired inference performance levels. Recently, various hardware accelerators for deep CNNs have been carried out to enhance the throughput of CNNs. Among these accelerators, field programmable gate array (FPGA)-based ones have gained a lot of interest due to their high performance, low power consumption, high reconfigurability, and fast development cycle. Furthermore, the availability of high-level synthesis (HLS) tools lowers the programming burden and increases the productivity of the FPGA-based accelerator designers. In this paper, a C++ HLS implementation for FPGA-based accelerator for the convolutional layers of CNNs is proposed. As a case study, we evaluate the proposed accelerator using Resnet50 CNN on Xilinx Zynq UltraScale+ MPSoC ZCU104 evaluation board using SDSoC development environment, achieving up to 339x inference speedup.**

***Index Terms*—Convolutional Neural Networks (CNNs), Field Programmable Gate Arrays (FPGAs), Hardware Accelerator, High-Level Synthesis (HLS), SDSoC.**

I. Introduction

Convolutional Neural Network (CNN) is one of the state-of-the-art deep learning architectures. It is constructed by adding a set of convolutional layers on top of the traditional Artificial Neural Network (ANN). With enough amount of labeled data, CNNs automatically learn and extract complex features. As a result, they provide high accuracy and are widely used in a variety of applications such as computer vision, speech recognition, and natural language processing. Furthermore, in recent years, incredible progress has been made, and state-of-the-art CNNs can currently surpass humans in image classification [1].

This outstanding performance of CNNs comes at a high cost in terms of computational complexity and resources due to the large number of parameters and required operations. Moreover, recent CNNs stack more layers to adapt with the increased complexity of the target applications and achieve higher accuracy. Therefore, for efficient processing, they require a scalable platform that offers massive parallelization opportunities in addition to large high-speed memory resources.

Central Processing Units (CPUs) have a fixed amount of resources and provide limited opportunities for parallel operations. Thus, CPUs fail to achieve the required performance levels. Alternatively, Graphical Processing Units (GPUs), which are designed for parallel processing, offer high throughputs due to their high memory bandwidth and highly parallel architectures that make them very efficient for floating-point matrix-based operations. However, GPUs have high power consumption numbers. Hence, they cannot be integrated into embedded platforms that are powered by batteries such as smartphones, drones, or wearable devices. Furthermore, these devices are limited not only in terms of power consumption but also in terms of physical size.

Therefore, many FPGA-based and ASIC-based CNN accelerators have been proposed recently [2]–[11]. Among these approaches, FPGA-based hardware accelerators have gained a lot of interest due to their high performance, low power consumption, high reconfigurability, and fast development cycle. Furthermore, the availability of high-level synthesis (HLS) tools reduces the programming barrier for FPGA-based accelerator designers, increasing their productivity [12], [13]. Moreover, Xilinx, an FPGA vendor, has introduced SDSoC, a hardware-software co-compiler that enables the implementation of heterogeneous FPGA-CPU platform. The fundamental characteristic of SDSoC is that it converts C++ design specifications into a hardware accelerator with minimal work and time, allowing for great performance while consuming minimal energy.

Convolution operations dominate the CNN operations. Thus, efficiently accelerating the convolutional layers is the key to achieve an efficient accelerator implementation. The convolution operation consists of many nested loops. By this means, it is required to effectively use loop optimization techniques such as loop interchange, loop tiling, and loop pipelining and unrolling [14]–[16]. These techniques optimize for the efficiency of the on-chip memory, data locality, and exploit the massively parallel architecture of the FPGA to achieve high throughput and low latency.

In this paper, we propose a C++ HLS implementation for FPGA-based accelerator to accelerate the convolution layers of CNNs. The rest of this paper is organized as follows. Section II gives an overview of High-Level Synthesis. Section III introduces the background of CNN. Section IV presents the details of the developed CNN accelerator architecture and the optimization techniques which have been used. We present the evaluation of our work in Section V, and the paper is

TABLE I
SHAPE PARAMETERS OF A CNN LAYER

Shape Parameter	Description
N_{if}	# of ifmap/filter channels
N_{of}	# of ofmap/3D filters
N_{iy}/N_{ix}	ifmap height/width
N_{oy}/N_{ox}	ofmap height/width
N_{ky}/N_{kx}	filter height/width
T_*	Tiling parameters
P^*	Parallelism parameters

concluded in Section VI.

II. HIGH-LEVEL SYNTHESIS

HLS is the transformation of a C specification into register transfer logic (RTL) implementation. It enables designers to work at a higher level of abstraction. Hence, improving the productivity by allowing them to focus on the algorithms being developed rather than the low-level hardware details. Further, HLS has more flexibility and programmability, and faster development and verification time than RTL [12], [13].

The HLS tools provide directives, also called pragmas, to control the synthesis process and optimize the design. Using these directives, a designer can easily explore the design space. Thus, HLS enables faster deployment and time to market. However, HLS has some challenges. The designer needs tool expertise and to think in hardware. Thus, a VLSI background is needed. Moreover, HLS is not only about pragmas insertion, many times the code needs to be refactored to be HLS-friendly and hardware-aware.

III. THE CNN

The CNN learning structure is constructed by adding a set of convolutional layers, and optionally pooling layers, on the top of the Fully Connected (FC) layers that construct the traditional Neural Network (NN). The added convolutional layers form the features extractor component of the CNN while the FC layers form the classifier part.

By this means, instead of feeding the raw input image tensor directly to the first FC layer in the CNN, we first apply filters to extract the features that distinguish one image from another. Thus, the convolutional layers narrow down the content of the input image tensor to emphasize specific and distinct details. As a result, the FC layers process a low-dimensional features tensor with more focused and presumably more accurate information. The FC layers determine which class the input image may belong to. The shape parameters of a CNN layer are listed in Table I.

A. The Convolution Layer

The convolution layer (Conv) convolves the input tensors, input feature maps (ifmaps) and kernels, to produce the output feature maps (ofmaps) tensor. For every pixel in the ifmaps tensor, it takes its value and the values of its neighbor pixels according to the kernel window. Then, it multiplies each pixel value by the corresponding value in the kernel window. The output pixel value is the accumulation of these values over the tensors channels. Then, the kernel window is shifted by the stride S to produce the next output pixel value. The computation of the convolution layer is illustrated using Fig. 1 and Fig. 2.

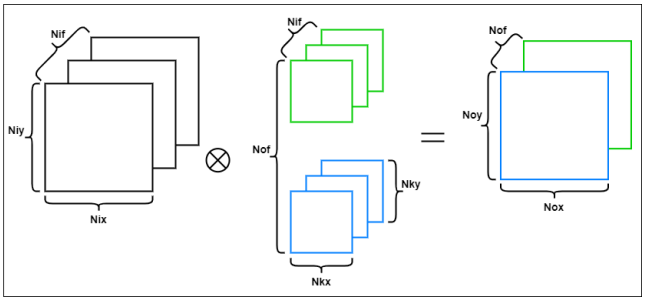

Fig. 1. An example for the convolution operation with an ifmaps tensor and weights tensor to produce the ofmaps tensor.

```
for(y=0; y < Noy; ++y)
for(x=0; x < Nox; ++x)
for(n=0; n < Nof; ++n)
for(i=0; i < Nif; ++i)
for(ky=0; ky < Nky; ++ky)
for(kx=0; kx < Nkx; ++kx)
  ofmaps[y][x][n] +=
    ifmaps[ky+y*S][kx+x*S][i] *
    weights[ky][kx][i][n];
```

Fig. 2. Pseudo code for the convolution operation (assuming the output buffer is initialized with the bias). 'S' is the convolution stride

IV. THE PROPOSED CNN ACCELERATOR

A. Accelerator Architecture

A System-on-Chip (SoC) platform with a processing system (PS) and Programmable Logic (PL) is used. The CNN model runs on the PS and uses the PL for acceleration. In our evaluations, when the CNN model is run completely on the PS, the convolutional layers dominate the inference time by more than 99%. Therefore, our work focuses on accelerating the convolution module by developing an efficient FPGA-based accelerator. The architecture of the developed accelerator is presented in Fig. 3.

The PS runs the entry function which allocates memory in the DRAM and initializes it with the target model's parameters and the input images. The Pooling module and the softmax activation function are run on the PS. The FC module is scheduled to run as a Conv. layer on the developed accelerator.

The Conv layer acceleration on the PL consists mainly of four key components: multiply and accumulations (MACs) array, on-chip buffers, external memory, and the AXI4 memory interfaces. The MAC operation is the core of the convolution operation. The developed accelerator has a uniform and scalable MACs array architecture that is reusable by all convolutional modules with the highest efficiency.

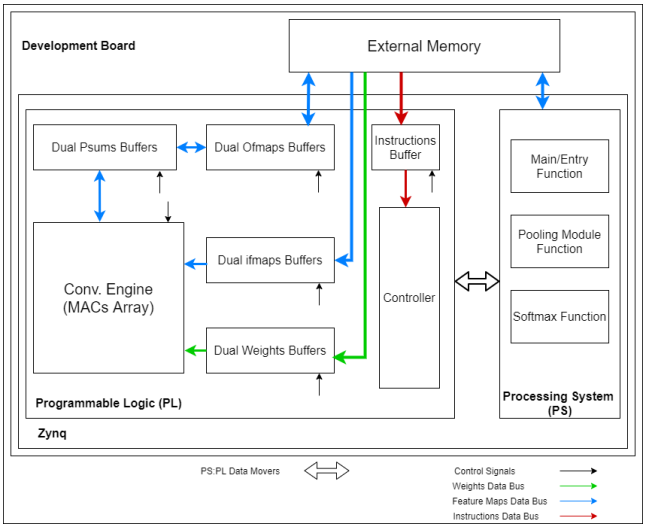

Fig. 3. Architecture of the developed accelerator.

```
for(it=0; it < ⌈Tif/Pif⌉; ++it)
for(ky=0; ky < Nky; ++ky)
for(kx=0; kx < Nkx; ++kx)
for(yt=0; yt < ⌈Toy/Poy⌉; ++yt)
for(xt=0; xt < ⌈Tox/Pox⌉; ++xt)
for(nt=0; nt < ⌈Tof/Pof⌉; ++nt)
 #pragma HLS PIPELINE
 for(y=0; y < Poy; ++y)
 for(x=0; x < Pox; ++x)
 for(i=0; i < Pif; ++i)
 for(n=0; n < Pof; ++n)
   psums[y][x][n] +=
     weightsTile[ky][kx][it*Pif + i][nt*
         Pof + n] *
     ifmapsTile[ky + (yt*Poy + y)*S][kx +
         (xt*Pox + x)*S][i];
```

Fig. 4. Pseudo code for the tiled convolution operation.

The on-chip buffers are used to cache data read from external memory in order to increase the data reuse and avoid rereading it from external memory. Moreover, the dual buffering technique is used to reduce the latency by hiding the data movements behind the computation time [8].

Because of the limited on-chip memory and the high number of parameters in current CNNs, the parameters of the target CNN and the ofmaps tensor of each layer are stored in external memory. Hence, the AXI4 master data movers serve as a link between the on-chip buffers and external memory. Furthermore, AXI4 burst mode data transfers are used for higher data throughput.

The ifmaps are padded, if needed, while loading them from external memory into the on-chip buffer. The CNN weights copied from external memory are stored in the weights buffers. The partial sums (psums) buffer is used to store the results of the MACs array.

The ofmaps buffer is constructed with a depth that equals the maximum depth of the ofmaps tensor of any Conv layer in the target CNN. The psums buffer results are cached in the ofmaps buffer. Hence, the external memory is not used as the psums cache. Furthermore, by using the ofmaps buffer instead of the psums buffer to communicate with the external memory, the MACs operations are not bounded by the external memory communication time.

B. Convolution Loops Optimization

1) Loop Tiling: Due to the limited on-chip memory, the input and output tensors of most layers don't fit on-chip. Thus, we must partition them into a set of tiles where each tile can fit in the available on-chip buffer [14]. Therefore, the tiling factors are driven mainly by the available on-chip memory on the target FPGA. The ifmaps tile buffer height and width must be big enough to cache the required number of rows and columns needed to produce the ofmaps tile rows and columns according to (1) and (2).

$$T_{iy} = N_{ky} + S * (T_{oy} - 1) \tag{1}$$

$$T_{ix} = N_{kx} + S * (T_{ox} - 1) \tag{2}$$

Also, in order to fully utilize the available accelerator parallelism resources, the tiling factors are set to be the same as the parallelism factors, i.e., $T_{oy} = P_{oy}$, $T_{ox} = P_{ox}$, and $T_{of} = P_{of}$.

2) Loop Transformation: The loop computations order affects the data reuse opportunities and the number of memory accesses. Further, the HLS tool cannot pipeline a loop that contains a loop with a variable bound. Thus, the loops are reordered such that they achieve the highest data reuse factors and the minimum data movements and memory accesses. Also, the loops with constant bounds are set to be the innermost loops. Fig. 4 illustrates the new loops computation order. The introduced loops order enables the reuse of the same kernel element $P_{oy} * P_{ox}$ times and the ifmaps pixel P_{of} times.

3) Loop Pipelining and Unrolling: Loop pipelining is one of the key optimization techniques to obtain a high performance design by allowing the concurrent execution of operations. Loop unrolling is employed to create multiple independent operations. This allows loop iterations to occur in parallel and utilizes the massively parallel architecture of the FPGA to achieve high throughput and low latency. As shown in Fig. 4, the pipeline pragma is inserted at the level of the innermost loop with a variable bound to pipeline it and automatically unrolls all the loops in the hierarchy below.

V. Evaluation

The proposed CNN accelerator is demonstrated by accelerating the inference process for ResNet50 [1] CNN on Xilinx Zynq UltraScale+ MPSoC ZCU104 evaluation board. This board has Xilinx Zynq ZU7EV chip with a PS that runs at 1.2 GHz.

TABLE II
INFERENCE TIMING ON PS

CNN	ResNet50
# of Operations (GOPs)	7.74
Conv Clock Cycles	$8.32\text{x}10^{10}$
Pooling Clock Cycles	$3.2\text{x}10^{7}$
Total Clock Cycles	$8.33\text{x}10^{10}$
Latency/Image (s)	69.4

TABLE III
ACCELERATION RESULTS

CNN	ResNet50	ResNet50
Data Precision	16-bit float	32-bit float
Clock (MHz)	150	150
$B_{iy}\text{x}B_{ix}\text{x}B_{if}$	10x10x2048	9x9x1024
$P_{oy}\text{x}P_{ox}\text{x}P_{if}\text{x}P_{of}$	1x1x8x64	1x1x4x64
DSPs	$1,591$ (92%)	$1,331$ (77%)
LUTs	$125,926$ (54.66%)	$120,531$ (52.3%)
REGs	$136,586$ (29.64%)	$162,168$ (35.2%)
BRAMs	188 (60.26%)	236 (75.64%)
URAMs	78 (81.25%)	90 (93.75%)
Power (Watt)	8	9.2
Conv Latency/Image (ms)	204.5	350.8
Latency/Image (ms)	239.8	378.3
Speedup to PS	339x	198x

We used Xilinx Vivado-HLS tool (v2019.1) to conduct pre-synthesis C simulation and C/RTL co-simulation. Also, it is used to get the pre-synthesis reports on the resources utilization estimates for design space exploration and performance estimation. Then, Xilinx SDSoC tool (v2019.1) is used to compile the PS functions and synthesize and place-and-route the accelerator functions on PL.

In our experiments, the benchmarking is conducted with a batch size of 1. Table II shows the breakdown of the inference on the PS and table III shows the detailed breakdown of the inference accelerated using PL in addition to the PL resources utilization. The resources utilization is reported after the place-and-route is completed. The developed accelerator utilizes an ifmaps buffer with dimensions of 9x9x1024 for the floating point implementation and 10x10x2048 for the half-floating point implementation. Compared to the PS-based inference, the developed accelerator achieves 198x and 339x speedup using floating-point and half floating-point data types, respectively.

VI. CONCLUSION

In this paper, HLS was utilized to build an FPGA-based inference accelerator for CNNs on a Xilinx SoC. The proposed work focused on speeding up the convolutional layers, which account for the majority of the inference time on the PS side of the SoC. Hence, loop optimization techniques such as loop tiling, loop interchange, loop pipelining, and loop unrolling are employed. The developed accelerator was synthesized using Xilinx SDSoC development environment. Then, it was benchmarked using a complex CNN, ResNet50, on Xilinx Zynq UltraScale+ MPSoC ZCU104 evaluation board. In comparison to the PS-based inference, the created accelerator achieved up to 339x inference speedup.

ACKNOWLEDGMENT

This work was partially funded by ONE Lab at Zewail City of Science and Technology and Cairo University, Siemens EDA (Mentor Graphics), ASRT, NTRA, and ITAC.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[2] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[3] A. Shawahna, S. M. Sait, and A. El-Maleh, "Fpga-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019.

[4] D. Gschwend, "Zynqnet: An fpga-accelerated embedded convolutional neural network," 2020.

[5] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, (New York, NY, USA), p. 161–170, Association for Computing Machinery, 2015.

[6] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2016.

[7] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and et al., "Fast inference of deep neural networks in fpgas for particle physics," *Journal of Instrumentation*, vol. 13, p. P07027–P07027, Jul 2018.

[8] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Automatic compilation of diverse cnns onto high-performance fpga accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 2, pp. 424–437, 2020.

[9] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[10] P. G. Mousouliotis and L. P. Petrou, "Squeezejet: High-level synthesis accelerator design for deep convolutional neural networks," *Lecture Notes in Computer Science*, p. 55–66, 2018.

[11] R. Osama and H. Mostafa, "Implementation of deep neural networks on fpga-cpu platform using xilinx sdsoc," *Springer Analog Integrated Circuits and Signal Processing*, vol. 106, pp. 399–408, 2021.

[12] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, "High-level synthesis: Productivity, performance, and software constraints," *JECE*, vol. 2012, Jan. 2012.

[13] M. Fingeroff, *High-Level Synthesis Blue Book*. Xlibris US, 2010.

[14] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," 2018.

[15] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, (New York, NY, USA), p. 45–54, Association for Computing Machinery, 2017.

[16] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing the convolution operation to accelerate deep neural networks on fpga," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 7, pp. 1354–1367, 2018.