Regular Article

An FPGA-based Convolution IP Core for Deep Neural Networks Acceleration

Xuan-Quang Nguyen[1,2], Cuong Pham-Quoc[1,2]

[1] Ho Chi Minh City University of Technology (HCMUT), Ho Chi Minh City, Vietnam
[2] Vietnam National University - Ho Chi Minh City, Vietnam

Correspondence: Cuong Pham-Quoc, cuongpham@hcmut.edu.vn

Communication: received 25 August 2021, revised 20 September 2021, accepted 21 September 2021
Online publication: 22 October 2021, Digital Object Identifier: 10.21553/rev-jec.286
The associate editor coordinating the review of this article and recommending it for publication was Prof. Tran Manh Ha.

Abstract– **The development of machine learning has made a revolution in various applications such as object detection, image/video recognition, and semantic segmentation. Neural networks, a class of machine learning, play a crucial role in this process because of their remarkable improvement over traditional algorithms. However, neural networks are now going deeper and cost a significant amount of computation operations. Therefore they usually work ineffectively in edge devices that have limited resources and low performance. In this paper, we research a solution to accelerate the neural network inference phase using FPGA-based platforms. We analyze neural network models, their mathematical operations, and the inference phase in various platforms. We also profile the characteristics that affect the performance of neural network inference. Based on the analysis, we propose an architecture to accelerate the convolution operation used in most neural networks and takes up most of the computations in networks in terms of parallelism, data reuse, and memory management. We conduct different experiments to validate the FPGA-based convolution core architecture as well as to compare performance. Experimental results show that the core is platform-independent. The core outperforms a quad-core ARM processor functioning at 1.2 GHz and a 6-core Intel CPU with speed-ups of up to 15.69× and 2.78×, respectively.**

Keywords– **Convolution neural network; Reconfigurable hardware; FPGA design.**

1 Introduction

Deep neural networks (DNNs) have achieved excellent accuracy over traditional algorithms in many applications. On the other hand, DNN models are going deeper and require more storage and computation complexity, demanding high-performance computing platforms to work efficiently. In edge computing deployments, these models are pre-trained in data centers, and the network inferences are then performed near data sources. However, most edge devices suffer from low performance and limited computation resources such as less storage and energy capacity [1].

To overcome this obstacle, we use FPGA-based hardware accelerator platforms for edge computing devices where we can exploit the computation flexibility of host processors as well as the high performance of reconfigurable fabrics [2, 3]. Furthermore, although FPGAs suffer from low working frequency, they outperform GPUs in energy consumption and provide higher performance and energy-efficient than CPUs [4]. Therefore, an FPGA-based hardware accelerator is a promising approach for building high-performance DNN in edge devices.

Many methods have been proposed to reduce the cost of network computation. From a software point of view, researchers developed lightweight models such as MobileNet [5], EfficientNet [6] that reduced the size of networks. Quantized versions of these models are also implemented to avoid the complexity of floating-point computation in hardware. From a hardware perspective, proposed optimization methods are about improving parallelism or buffering a high amount of data to avoid many secondary memory accesses. There are many studies on hardware acceleration for DNNs. That is one of the most concerning problems recently. Convolution engines which are based on fast algorithms are proposed in [7], [8] while [9] and [10] reduced bit-width of neural network parameters to decrease computation and storage cost. In [11], authors proposed look-up table solution for fast operations. In a wider consideration, [12] showed that research topics on FPGA-based neural network accelerator usually are optimizing models for effective computation and storage utilization, developing computational units, architecture for parallelism.

Most DNN models mainly use convolution and matrix multiplication. These two mathematical operations require the multiply and accumulate (MAC) as the basic computation unit. Because of the out-of-order for cumulative multiplication in the two primary network operations, we can perform the MAC operation in parallel to archive higher throughput and lower latency. However, we cannot calculate all needed MAC operations of a neural network's layer simultaneously because the input feature map and the layer's parameters could be huge to store into edge devices' on-chip memory. As a result, edge devices need extra off-chip

memory. In this situation, read/write operations to the off-chip memory must be optimized to maximize the bandwidth. Fortunately, the computation can be performed in any order as mentioned above. Therefore, we can re-order the MAC operations to get high data reuse. With this method, a part of parameters can be kept in on-chip memory for computing while still used. Then, they are replaced by the following parameters and never be used again.

In this paper, we propose an FPGA-based architecture to accelerate the convolution operations in edge devices. We aim to design and develop a general convolution IP core that can be used for different DNN models on FPGA platforms instead of optimization for a particular purpose or platform as research mentioned above. Our approach is a software/hardware co-design to perform the convolution on any FPGA-based system-on-chip platform, which is suitable for inference in edge devices. In other words, we propose an architecture that is platform-independent for convolution accelerator on edge devices. To validate the proposed Convolution IP architecture, we develop a synthesizable hardware core with Verilog-HDL. The core is then synthesized and implemented on the Xilinx Zynq UltraScale+ platform [13]. Experimental results show that the implemented hardware IP core archives 2.78× faster than performing the same convolution on the CPU Intel core i7-9750H and 15.69× faster than processing on the ARM Cortex-A53 software only.

The rest of the paper is organized as follows. First, Section 2 presents an overview of convolution computation as well as optimization techniques to improve performance. Second, we present our proposed Convolution IP core architecture in Section 3. Implementation of the proposed IP core architecture with Verilog-HDL on the FPGA platform as mentioned above is discussed in Section 4. Third, we conduct several experiments and present results in Section 5. Finally, we conclude our work in Section 6.

2 Background & Optimization Techniques

In this section, we present an overview of convolution computation. Based on that, we design our IP core architecture. Optimization techniques, including temporal and spatial data reuse, are introduced. These techniques are also applied to our IP core to improve performance.

2.1 Convolution Computation

A deep neural network or DNN is the name of a neural network with greater than three layers. DNN models in recent studies have fully connected layers with multiple layers inside and many convolution layers in their structure. In convolution layers, the primary computation is high-dimensional convolution. The input feature maps (I) are constructed as a 3D matrix ($fmaps = [H \times W \times C]$) in which the shape notations are height (H), width (W), and number of channels (C). For example, a bitmap image with the resolution of 256×256 pixels and 3 color channels (red, green, and blue) per pixel can be represented by $fmaps = [256 \times 256 \times 3]$ with $H = 256$, $W = 256$, and $C = 3$. The weights are also structured as 3D filter matrices (F, also called kernels), where the dimensions are height (R), width (S), and the number of channels (C). The number of kernels (so-called *depth*) used for a particular DNN model depends on the number of features in the input we would like to recognize. Assume that we apply multiple filters ($depth = M$) to input feature maps to generate M output channels of the output feature maps (3D matrix $O = [P \times Q \times M]$) in which the dimensions are height (P), width (Q), and the number of channels (M). Figure 1 illustrates a convolution computation of C channels input $fmaps$ and M filters. In this paper, white blocks illustrate inputs, green rectangles/squares represent kernels, and blue ones depict outputs.

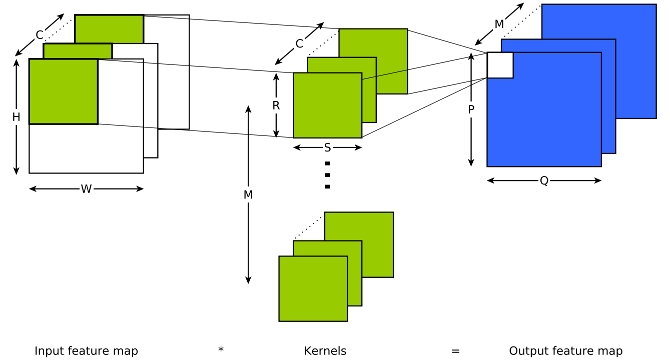


Figure 1. Convolution computation.

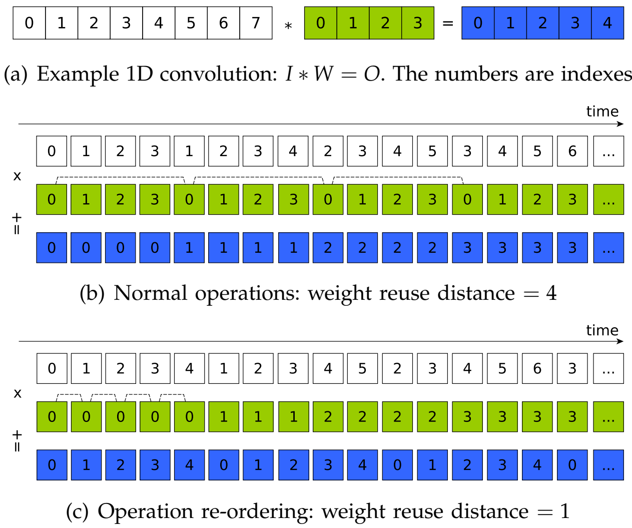
The operations of the convolution computation shown in Figure 1 is defined in Equation (1)

$$O(p,q,m) = \sum_{c,r,s} I[(Up+r),(Uq+s),c] * F(c,r,s,m), \quad (1)$$

where m, p, q, c, r, and s are indexes corresponding to M, P, Q, C, R, and S dimensions respectively and U is the stride - number of skipped intermediate locations when moving filters for each computation. Each output point results from a dot product of elements in input activations and filter weights across the index c, r, and s. This operation is MAC and is performed in an arithmetic unit called *processing element* (PE). The execution order of MAC operations is not important. This results in two strategies (temporal and spatial data reuse) of data reuse to achieve higher parallelism computation and reduce data communication overhead. The next subsections present the two approaches for optimizing the performance of the convolution computation on hardware.

2.2 Temporal Data Reuse

When the same data value is used multiple times by one PE, temporal reuse should be performed [14]. The temporal reuse method re-orders the MAC operations to keep data in PEs locally so that the number of memory reads for the value can be reduced dramatically. The time interval between two consecutive uses of the



(a) Example 1D convolution: $I * W = O$. The numbers are indexes

(b) Normal operations: weight reuse distance $= 4$

(c) Operation re-ordering: weight reuse distance $= 1$

Figure 2. An example applying re-ordering in 1D convolution for temporal reuse.

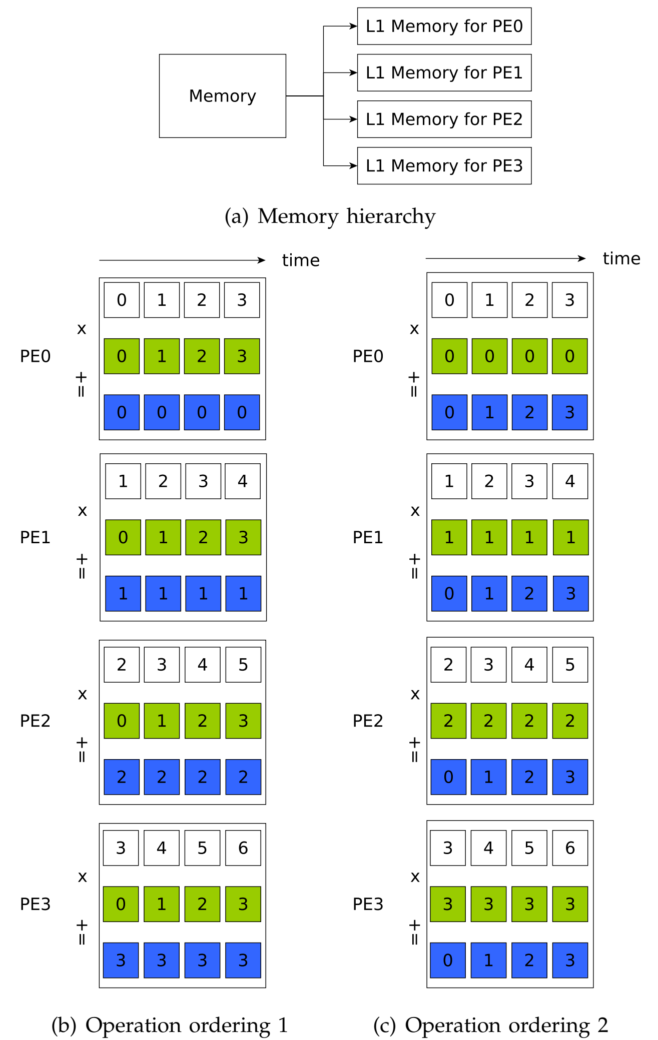
value is called the *reuse distance* (*rd*). In this term, the philosophy is to minimize the reuse distance. Based on [14], Figure 2 describes the phenomenon of temporal data reuse.

In the example shown in Figure 2, we simplify the case by performing 1D convolution. In this case, the convolution of an 1×8 input (white vector) and an 1×4 weight (green vector) produces an 1×5 output (blue vector) with indices as shown in Figure 2(a). Conventionally, as illustrated in Figure 2(b), we calculate each point of output in the sequential order. This order causes the reuse distance of weight to be $rd = 4$. Mapping to hardware perspective, if the weight is stored in PE for performing MAC operations, the weight values need to fetch time by time. Hence, this approach costs a large amount of data movement and memory access. After re-ordering, as depicted in Figure 2(c), the reuse distance is reduced to $rd = 1$. Therefore, we can buffer a weight value in a PE to use in four consecutive operations before replacing it with a new value to minimize the number of memory accesses when fetching weight values.

2.3 Spatial Data Reuse

Spatial reuse is the use of the same values for multiple PEs simultaneously. This method improves the parallelism level of the computation. Due to data independence, MAC operations with the same weight values can be performed in parallel (with different inputs). Although the space limitation (resources in hardware perspective) can prevent the spatial reuse for all operations, we can take advantage of the memory hierarchy by using both reuse strategies. We take the 1D convolution example mentioned above to explore the spatial data reuse technique. Figure 3 presents the spatial data reuse technique.

Assume we have four PEs and a memory hierarchy as depicted in Figure 3(a); others are not shown to simplify. Figure 3(b) depicts the way upon which the spatial data reuse applies to the model presented



(a) Memory hierarchy

(b) Operation ordering 1

(c) Operation ordering 2

Figure 3. An example applying re-ordering in 1-D convolution for spatial reuse.

Figure 2(b). While PE0 is calculating convolution of the kernel and four first input values (index 0, 1, 2, and 3), PE1 is computing for the next input stride (index 1, 2, 3, and 4). Compared to Figure 2(b), this computational model needs only four cycles to generate four convolution results instead of 16 as the temporal data reuse model. However, this computational model can be further optimized by exploiting both temporal and spatial data reuse techniques as depicted in Figure 3(c). For example, PE0 calculates convolution for the kernel's first value while PE1, PE2, and PE3 compute the second, third, and fourth values. Compared to the previous model, we improve parallelism by applying the spatial data reuse approach and reducing external memory accesses by exploiting the temporal data reuse technique.

3 FPGA-based Convolution IP Architecture

In this section, we introduce the proposed architecture for the FPGA-based convolution computation IP core. The proposed architecture is platform-independent so that there is no FPGA family considered in this section.

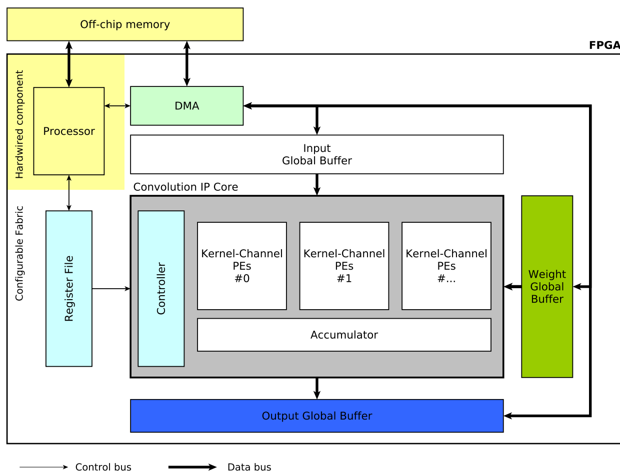


Figure 4. Overview architecture of the acceleration of convolution computation.

3.1 Overview Architecture

By applying the data optimization techniques mentioned above and adapting to the system-on-chip platform architecture in general, we propose a novel architecture to accelerate the convolution for edge devices as depicted in Figure 4.

In this architecture, a *Processor*, which usually is hardwired in SoC chips, takes responsibility for controlling the entire system and doing other steps for DNN-based applications. Firstly, it fetches data and model parameters to its memory (the *Off-chip memory* block in the figure) from the secondary memory (usually from an SD card). When starting the convolution computation, these data have to be delivered to buffers (including *Input Global Buffer* and *Weight Global Buffer*) and *Register File* in the Configurable Fabric for the processing of the *Convolution IP Core* (functioning as a hardware accelerator). The extensive data cannot be transferred to the buffers by the host processor because of the low throughput. Therefore, a direct memory access module (*DMA*) takes this responsibility. It transfers data that the host processor loads from the off-chip memory (usually a DDR) to the on-chip memory, the Block RAMs memory inside FPGA devices.

Along with the two buffers mentioned above, Input Global and Weight Global, the *Output Global Buffer* is needed to store outputs generated by the core. The Input Global and Weight Global buffers store the input feature maps and weight data, respectively. Because of the flexibility and variance of DNN models, parameters of the accelerator, including sizes of input maps (H, W, and C), dimensions of kernels (R, S, and C) as well as kernel depth (M) should be configured by the Processor via *Register File* to work correctly. The Convolution IP Core block executes the primary operations of the convolution computation. Inside the core, there is an array of *Kernel-Channel Processing Engine* (KCPE) for calculating convolutions of inputs and kernels. The number of KCPE elements depends on the number of resources available in target FPGA devices. More details of these elements are discussed in the next section.

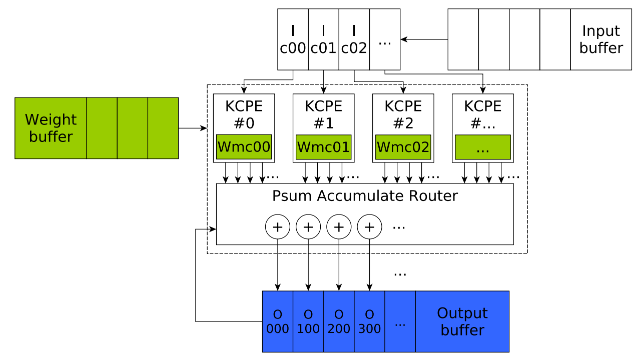


Figure 5. Architecture of the Convolution IP core.

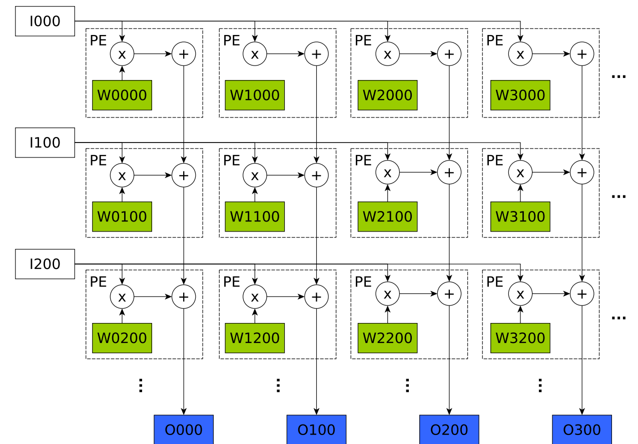


Figure 6. Kernel-Channel Processing Engine.

3.2 Convolution IP core

Figure 5 presents in detail the architecture of the Convolution IP core. When analyzing deeper into the core architecture, the accelerator core contains an array of KCPEs. Each KCPE can produce partial sums of multiple kernels and channels at the same time.

The principle of dataflow is weight stationery in which a part of weight data of some kernels are loaded to KCPEs at a time and kept locally for calculating the partial sum of all output elements before replaced by the next weight data part. With this architecture, the core can perform MAC operations for multiple output elements simultaneously. The number of output elements calculated in parallel depends on the number of KCPEs. Furthermore, the core can execute computation for multiple kernels and channels simultaneously. The number of kernels and channels depend on the number of *Processing Elements* (PEs) contained in each KCPE. Because each KCPE can calculate the convolution of multiple kernels, the array of KCPEs produces partial sums of different output channels. The *Psum Accumulate Router* component adds and drives these partial sums to the proper output addresses. After computed, output data should be sent back to the off-chip memory to free the output buffer for the subsequent computations.

3.3 Kernel-Channel Processing Engine

Figure 6 depicts the micro-architecture of a Kernel-Channel Processing Engine. Each KCPE processes the

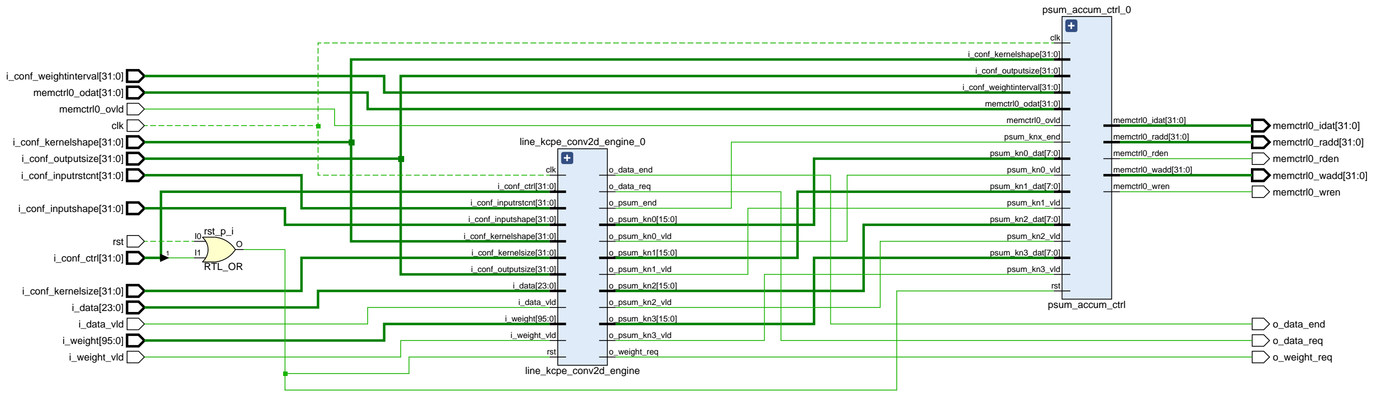


Figure 7. Schematic of a Convolution IP core.

dot product of an input feature map element and a respective weight element of multiple channels and kernels. The results of engines are accumulated and written to the Output Global buffer. These KCPEs contain a matrix of Processing Elements that perform MACs. Refer to the NVDLA [15], Google's TPU [16] weight static dataflow, the Kernel-Channel Processing Engine processes a weight from input channels and output channels every cycle.

The number of KCPEs in a Convolution IP core should be optimized according to the characteristics of a specific model and the available resources in target hardware platforms. For example, assume that a Convolution core can process four kernels and three channels per kernel (as illustrated in Figure 6). The weight values of the three channels of 4 kernels data are read from the Weight Global buffer to local registers inside the PEs ($W0100$, $W1000$,...). As soon as the weights are loaded, input feature map data are streamed from the Input Global buffer through KCPEs and multicast to all columns of the PEs matrix. Results of the process are partial sums of output elements that corresponding to output channels.

4 System Implementation

In this section, we present our implementation for the proposed Convolution IP core. The implementation is platform-independent for multiple FPGA platforms. We then develop a computation system that uses the core and the Zynq UltraScale+ MPSoC FPGA platform to validate the core. Besides, in this section, we also introduce a performance evaluation model to estimate the execution time of the Convolution IP core according to different input parameters.

4.1 Convolution Core Implementation

We use Verilog-HDL to describe the core manually to develop a platform-independent and synthesizable Convolution IP core for FPGA devices. Parameters are used to define the dimensions input map features (H, W, and C) and kernels characteristics (R, S, C, and M). Hence, the core can be used in different models as well as with various FPGA families. Figure 7 illustrates the schematic of the core that is generated automatically by Vivado with a particular set of parameter values.

4.2 Performance Model

Theoretically, the time interval to calculate a convolution computation of the core depends on both (1) the number of PEs which determines the number of MAC operations performed in parallel; and (2) the size of input feature maps and weight data. Assume that we compute a convolution of input $[H \times W \times C]$ and M filters $[R \times S \times C]$ which results in the output $[P \times Q \times M]$ in a core that consists of E KPCEs. Each KCPE accommodates K (kernels) $\times$ J (channels) PEs. The operating clock frequency of the core is f (Hz). The time to process the computation of the core can be estimated by Equation (2)

$$t_{\text{core}} = \left(\frac{H \times W \times C \times M \times R \times S}{E \times K \times J} + n_{\text{delay}}\right) \times \frac{1}{f}(s), \quad (2)$$

where n_{delay} is the number of cycles to calculate the multiplication in a PE. Equation 2 ignores the time to read data from the input and Weight Global buffers and to write data to the Output Global buffer. These time intervals depend on particular implementations. Besides, when input feature maps, weights, or output data are more extensive than buffer capacity, we should include the time to transfer data between off-chip memory and buffers. In this case, we have to move output data to the off-chip memory and fetch new data from it, then perform the new computation sequence. After all, the computation time should be the Equation (3).

$$t = t_{\text{core}} + t_{\text{buffer}} \times (n_{\text{overflow}} - 1) + n_{\text{overflow}} \times t_{\text{mem}}, \quad (3)$$

where t_{buffer} is the duration to access the global buffers (only at first read input data and weight and last write output). t_{mem} is the time to access the off-chip memory. Finally, n_{overflow} is the number of times that the amount calculated output data exceeds the output buffer capacity.

4.3 FPGA-based MPSoC Platform Implementation

Upon the convolution core, we implement a hardware accelerator system for convolution computation on the Ultra96v2 board [17] with the Zynq UltraScale+

MPSoC FPGA platform [13]. The system is built based on the overview architecture depicted in Figure 5 in which the hardwired quad-core Arm Cortex-A53 functions as the host processor to handle operations of the core as well as the entire system. Due to resources limitation, the system includes 3 KCPEs ($E = 3$), each of which contains 12 PEs for performing MACs of 4 $3 \times 3 \times 3$ kernels ($R = 3$, $S = 3$, $J = 3$, and $K = 4$) at a time. The DMA module is the Xilinx Central Data Memory Access (CDMA) for memory-mapped data transfer from a 2GB DDR off-chip memory. With the support of the PYNQ framework [18], we implement the software part, including DMA, accelerator engine drivers in Python. We use MAC cores of 8-bit integer computation in the edge deployment perspective to adapt quantized models originally designed for edge device inference.

5 Experiments

In this section, we present our experiments with different scenarios, including (1) processing convolution of only the host processor (the ARM processor); (2) processing convolution of only a Intel Core i7 processor; and (3) accelerating the host processor with our IP core; We also analyze our experimental results in this section.

5.1 Experimental Setup

To validate soundness and correctness of the Convolution IP core as well as compare performance, we conduct three different experiments with the same inputs as follows.

(i) *Software with ARM* (SWArm): only the ARM processor executes the entire program. In other words, the configurable fabrics are not used.

(ii) *Software with Intel* (SWInt): we use an Intel Core-i7-9750H with 6 physical cores and 12 threads, functioning at 2.6 GHz, to execute the program. With the support of the PyTorch library, all the threads are used to compute the convolution.

(iii) *Hardware accelerator* (HWAcc): in this experiment, the embedded ARM processor, hardwired in the PS part of the FPGA device functioning at 1.2 GHz, executes software part to manage data movement and handle the processing of modules in the configurable fabrics (PL part of the FPGA device) while the Convolution IP core, functioning at 300 MHz, processes all convolution computations. This is the ultimate goal of this paper, accelerating software processing by our IP core.

In these experiments, we use a color images with resolutions of 224×224 pixels with 3 color channels (red, green, and blue) as input maps. In other words, the input maps parameters are $fmaps = [224 \times 224 \times 3]$ (i.e., $H = 224$, $W = 224$, and $C = 3$). Three different kernels sets are used for these experiments including 4, 8, and 16 kernels (i.e., M is value of 4, 8, and 16, respectively). Each of kernels is a $[3 \times 3 \times 3]$ 3D matrix (i.e., $R = 3$ and $S = 3$). Please note that the core consists of 3 KCPEs ($E = 3$). Each KCPE accommodates 4 3×3 kernels (i.e., $K = 4$). Each kernel consists 3 channels (i.e., $J = 3$).

Table I
Synthesis Results for Our Convolution IP Core

FPGA device	#LUTs	#FFs	Max Frequency
xczu3egsbva484-1	1355	2159	476 MHz
	1.92 %	1.53 %	
7z020clg400-1	1372	2159	173 MHz
	2.58 %	2.03 %	
7vx690tffg1761-2	1372	2159	344 MHz
	0.32 %	0.25 %	
7a12tcpg238-3	1372	2159	243 Mhz
	17.15 %	13.49 %	

5.2 Experimental Results

In this section, we present results of the aforementioned experiments. At first we analyze synthesis and simulation results of the core. We then compared performance of the three experiments.

5.2.1 Synthesis results: The Convolution IP Core implementation are synthesized with various of Xilinx FPGA devices. Resources usages as well as maximum frequency for the core with each device are reported in Table I.

According to the table, the core uses less amount of hardware resources when synthesized alone. However, for the entire hardware accelerator system, depicted in Section 4.3, we need more additional resources due to other modules used such as DMA, Registers file, Buffers, and the AXI bus for system interconnect. The MPSoC hardware accelerator system with 256 KB for input buffer, 32 KB for weight buffer and 512 KB for output buffer used 93% available BRAMs of the FPGA but just 29% LUTs and 19% CLBs as flip-flop of the chip. This programmable logic was optimized to run in 300 MHz with 3.534 W total on-chip power.

5.2.2 Simulation results: To validate the correctness of the core, we simulate the RTL design of the core with ModelSim. A part of simulation waveform is shown in Figure 8. A careful analysis proves the correction of the core. We then further analyze the operation and realize that there are some unnecessary computations. These can be improved to achieve higher performance. For example, we do not bypass the MAC operations at edge of input tensor, or the zero multiplications should be recognized to reduce number of computations. Those are our future work to further optimize the core.

5.2.3 Performance analysis: In our implementation, we have $E = 3$, $K = 4$, $J = 3$, $n_{\text{delay}} = 4$ cycles and $f = 300 \times 10^6$ Hz. As mentioned above, we conduct experiments with input $I[224 \times 224 \times 3]$ with 4, 8, and 16 kernels $W[3 \times 3 \times 3]$. Experimental results show that average processing times of the core are 0.00056 s, 0.00107 s, and 0.00459 s, respectively. The details execution times of different experiments are summarized in Table II.

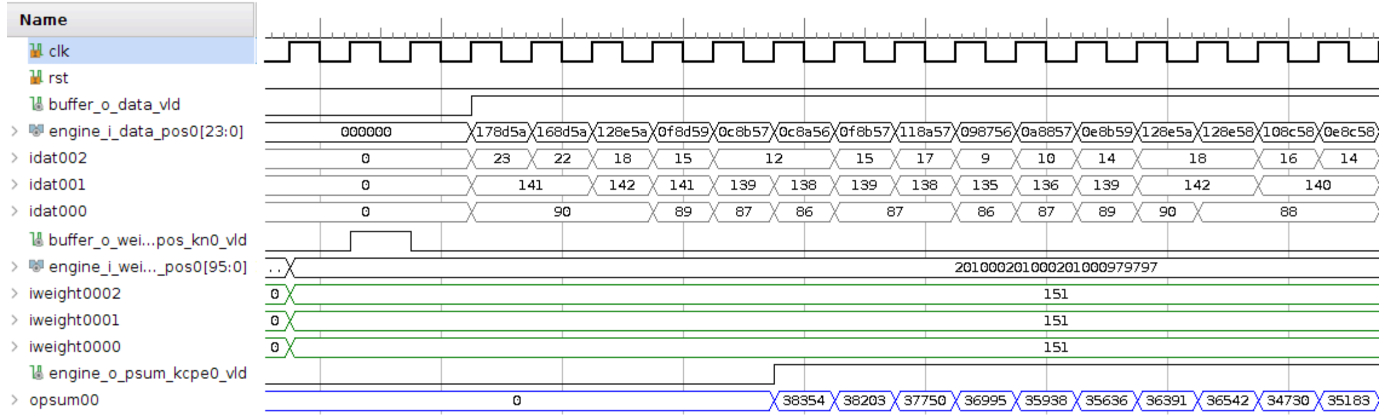


Figure 8. Simulation waveform of implemented convolution core.

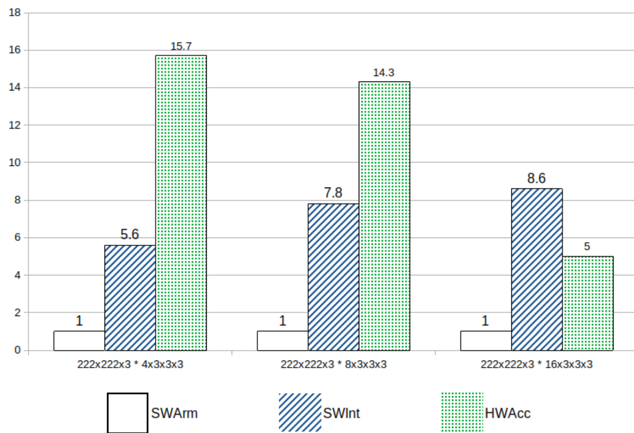


Figure 9. Speed ups comparison of our proposed core (HWAcc), software on ARM (SWArm), and software on Intel CPU (SWInt).

Table II
EXECUTION TIME (SECOND) OF THE THREE EXPERIMENTS WITH DIFFERENT KERNELS

# Kernels (M)	SWArm	SWInt	HWAcc
4	0.00879	0.00156	0.00056
8	0.01525	0.00196	0.00107
16	0.02312	0.00270	0.00459

Figure 9 depicts the speed-up of our SWInt and our core when compared to SWArm. According to the table and the figure, with moderate number of kernels (M is 4 or 8), our core outperforms both the ARM processor functioning at 1.2 GHz and the Intel processor functioning at 2.6 GHz when obtaining a speedup of up to 2.78× compared to the Intel processor and a speedup of up to 15.69× compared to the ARM processor. However, when the number of kernels becomes 16 ($M = 16$), the amount of outputs exceeds the Output Global size (exceeding 512 KB). In this case, the Intel CPU outperforms our core since our system needs to speed time for taking care the oversize data output.

6 Conclusion

In summary, we analysed DNN models and their mathematical computation and architecture of edge devices to summarize the bottleneck points of performing network inference at edge. This paper proposed an architecture for convolution accelerator on FPGA-based SoC platforms. The work also designs an accelerate core called kernel-channel processing engine that can process convolution on multiple input feature map channels and multiple kernels based on weight stationary dataflow principle. The combination of engines as an array helps the core perform computations on many input elements in parallel. From that point, we can reduce latency and increase throughput of computation. The implemented accelerator is about 2.78× faster than the Intel Core i7-9750H CPU and 15.69× faster than ARM Cortex-A53 with the same convolution computation in our experiments.

Acknowledgment

This research is funded by Vietnam National University - Ho Chi Minh City (VNU-HCM) under grant number B2021-20-02.

References

[1] R. Wu, X. Guo, J. Du, and J. Li, "Accelerating neural network inference on FPGA-Based platforms–a survey," *Electronics*, vol. 10, no. 9, p. 1025, 2021.

[2] C. Pham-Quoc, B. Kieu-Do-Nguyen, and T. Ngoc Thinh, "An FPGA-Based Seed Extension IP Core for BWA-MEM DNA Alignment," in *Proceedings of the International Conference on Advanced Computing and Applications (ACOMP)*, 2018, pp. 1–6.

[3] C. Pham-Quoc, B. Kieu-Do, and T. N. Thinh, "A high-performance FPGA-based BWA-MEM DNA sequence alignment," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 2, p. e5328, 2021, e5328 cpe.5328.

[4] C. Pham-Quoc, J. Heisswolf, S. Werner, Z. Al-Ars, J. Becker, and K. Bertels, "Hybrid interconnect design for heterogeneous hardware accelerators," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 843–846.

[5] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv preprint arXiv:1704.04861*, 2017.
[6] M. Tan and Q. V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," in *Proceedings of the International conference on machine learning*, 2019, pp. 6105–6114.
[7] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Proceedings of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 101–108.
[8] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," in *Proceedings of the 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–9.
[9] A. Podili, C. Zhang, and V. Prasanna, "Fast and efficient implementation of Convolutional Neural Networks on FPGA," in *Proceedings of the IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2017, pp. 11–18.
[10] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 75–84.
[11] M. Samragh, M. Ghasemzadeh, and F. Koushanfar, "Customizing Neural Networks for Efficient FPGA Implementation," in *Proceedings of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 85–92.
[12] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[DL] A Survey of FPGA-Based Neural Network Inference Accelerators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 1, Mar. 2019.
[13] Xilinx. (2021) Zynq ultrascale+ mpsoc. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html
[14] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *arXiv preprint arXiv:1703.09039*, 2017.
[15] Nvidia. (2017) Nvidia deep learning accelerator open source project. [Online]. Available: http://nvdla.org/
[16] N. Jouppi, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, C. Young, T. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. Ho, D. Hogberg, J. Hu, and N. Boden, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 6 2017, pp. 1–12.
[17] Avnet. (2021) Ultra96-V2 Board - Arm-based, Xilinx Zynq UltraScale+ MPSoC development board based on the Linaro 96Boards Consumer Edition specification. [Online]. Available: https://www.avnet.com/wps/portal/us/products/new-product-introductions/npi/aes-ultra96-v2/
[18] Xilinx. (2021) Python productivity for zynq. [Online]. Available: http://www.pynq.io/

Xuan-Quang Nguyen is currently a researcher at Computer Engineering Laboratory, Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology. He received his BEng and MEng from Ho Chi Minh City University of Technology, VNU-HCM, Vietnam in 2018 and 2021 respectively. His research interests include hardware accelerator, system-on-a-chip, and VLSI design.

Cuong Pham-Quoc received the BEng degree in 2007 and the MEng degree in 2009, both from the Faculty of Computer Science and Engineering, the Ho Chi Minh City University of Technology (HCMUT). He got his Ph.D. degree from the Computer Engineering Lab of the Delft University of Technology, the Netherlands. Currently, his work focuses on addressing the bottlenecks in FPGA-based designs for high-performance computing systems and proposing IoT-based solutions for issues in smart cities.