FPGA-Based Neural Network Acceleration for Handwritten Digit Recognition

Guobin Shen[1], Jindong Li[1], Zhi Zhou[2], and Xiang Chen[1(✉)]

[1] School of Electronics and Information Technology, Sun Yat-sen University, Guangzhou 510006, China
shengb3@mail2.sysu.edu.cn, lijd27@mail2.sysu.edu.cn, chenxiang@mail.sysu.edu.cn

[2] School of Data and Computer Science, Sun Yat-sen University, Guangzhou 510006, China
hustzhouzhi@gmail.com

Abstract. Convolutional neural network (CNN) has been widely employed in different engineering fields, as it achieves high performance for enormous applications. However, neural networks are computationally expensive and require extensive memory resource. While still implementing convolutional neural network using relatively few resources but achieving high computation speed has been an active research. In this paper, we propose an FPGA-based handwritten digit recognition acceleration method, applying the Lenet-5 model to the FPGA using Vivado High-Level Synthesis. By using fixed point quantization method, removing data dependencies and applying appropriate pipelining, the accuracy rate reaches 97.6% on MNIST dataset. On Zedboard, we achieve 3.65 times faster than running only on the Processing System (PS) of the same hardware.

Keywords: Convolutional neural network (CNN) · High-Level Synthesis (HLS) · Acceleration · FPGA

1 Introduction

Convolutional Neural Network (CNN) has brought breakthroughs in engineering fields such as signal processing, computer vision and robotics. In recent years, lots of influential Convolutional Neural Network structures have been proposed

This work was supported in part by the State's Key Project of Research and Development Plan under Grants 2019YFE0196400, in part by Industry-University Collaborative Education Program between SYSU and Digilent Technology: Edge AI Oriented Open Source Software and Hardware Makerspace, in part by the Guangdong R&D Project in Key Areas under Grant 2019B010158001, and in part by the Guangdong Provincial Special Fund For Modern Agriculture Industry Technology Innovation Teams under Grant 2020KJ122.

B. Li et al. (Eds.): IoTaaS 2020, LNICST 346, pp. 182–195, 2021.
https://doi.org/10.1007/978-3-030-67514-1_15

[5,6,9,11]. In 1998, Lenet-5 [10] structure proposed by Yann LeCun, achieved an astonishing accuracy of 98% in handwritten digit recognition. In 2012, Alexnet [11], designed by Alex Krizhevsky, significantly improved on the best performance in the image recognition. VGG-16 [9], Google-Net [10] and ResNet [5], also promoted the development of the neural network. However, as the network structure becomes more and more complicated, more computational resources are required. FPGA, a powerful hardware-acceleration architecture, can be used to speed up the computing. Recent study done by Chen et al. [12] presented a CNN accelerator achieving a peak performance of 61.62 GFLOPS under 100MHz working frequency, which greatly improves the traditional GPU accelerator. Lots of research have shown that FPGA has great advantages which still need to be exploited.

When compared with traditional processor architectures, the structures of the FPGA enable a high degree of parallelism. A processor executes a program as a sequence of instructions generated by processor compiler tools, which transforms an algorithm expressed in high level languages into assembly language. While an FPGA converts the program written in Hardware Description Languages (HDL) to a Register-Transfer Level (RTL), it reconfigures the integrated circuits inside the FPGA to best fit the design we need. This architecture can achieve a performance much better than the traditional processor architecture.

However, despite the great advantage of the FPGA architecture, designing a large-scale complex project using low-level hardware description languages is high-cost and inefficient. Figure 1 [2] illustrates the implementation time and achievable performance for different computation platforms. Development time of the FPGA using RTL is quite high, therefore this method has not been widely used.

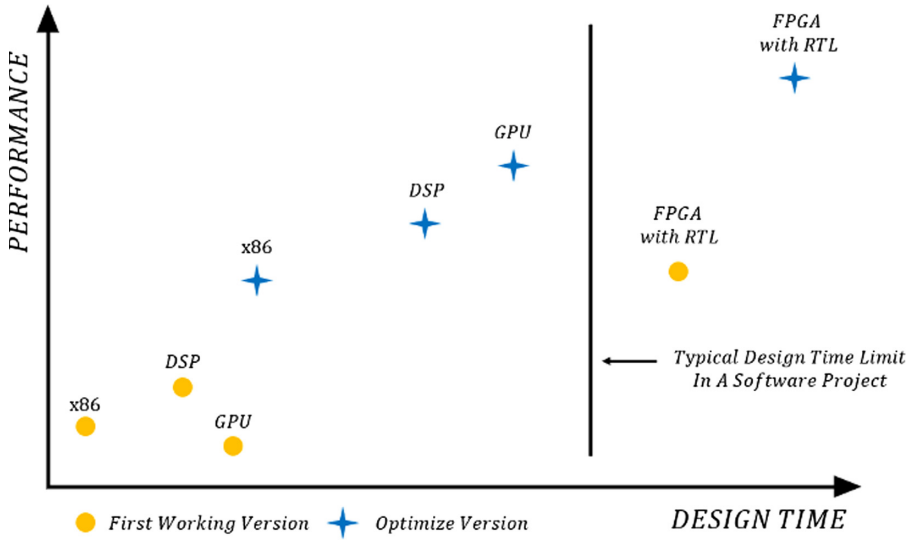

Fig. 1. Implementation time and achievable performance for different computation platforms.

Recently, Xilinx presented the High-Level Synthesis tool, it can automatically convert the high-level language C/C++ to the RTL model and HDL, which remove most of the difference in programing models between a processor and an FPGA. Without degrading the performance achieved by FPGA RTL design flow, FPGA with High-Level Synthesis (HLS) shortens the design time in a software project dramatically. Figure 2 [2] compares the result of the HLS design solution against other processor solutions.

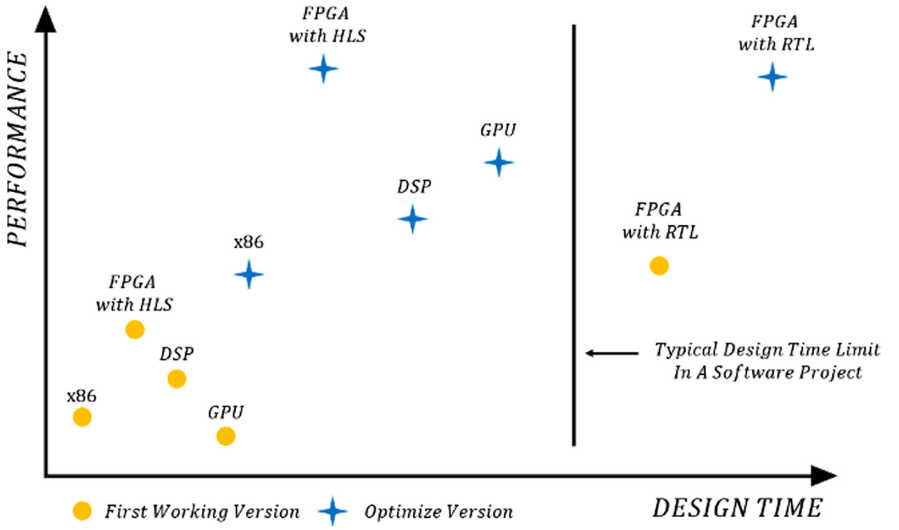

Fig. 2. Result of the HLS design solution against other processor solutions.

2 Related Work

2.1 LeNet-5

The LeNet [10] neural network was proposed by Yan LeCun, the father of Convolutional Neural Networks (CNN). LeNet is mainly used for the recognition

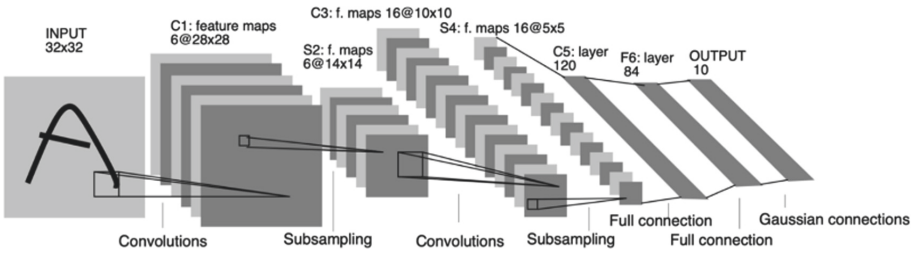

Fig. 3. Network structure of Lenet-5.

and classification of handwritten characters, and has been put into use in American banks. The implementation of LeNet has established the structure of CNN, and now much of the content in the neural network can be seen in the network structure of LeNet. Figure 3 shows the Lenet-5 network structure.

Layer1 is a convolution layer with 6 convolution kernels of 5×5, and the size of output feature is 28×28; Layer2 is the pooling layer that outputs 6 feature graphs of size 14×14. We use average pooling instead of the max pooling. Layer 3 is again a convolution layer with 16 convolution kernels of 5×5. Layer4 is similar to layer2, with size of 2×2 and output of 16 feature graphs. Layer5 is a convolution layer with 120 convolution kernels of size 5×5, and the output size of layer5 is 1×1. Then the layer5 is flatten and connects to the fully connected layer6. Using a 120×84 weight matrix, an output vector of size 84 is computed, then again using an 84×10 weight matrix, we can get the final output feature vector of size 10.

2.2 Design Flow of the High-Level Synthesis

Figure 4 shows the workflow for the FPGA development of Lenet-5 using Vivado HLS.

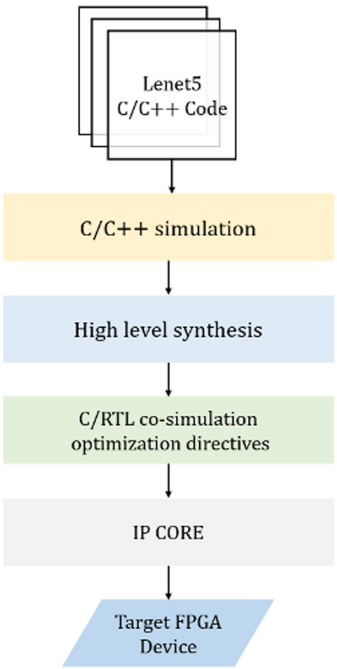

Fig. 4. Workflow for the FPGA development of HLS.

In our work, we used C/C++ as the development language, which is supported by the Vivado HLS. All the kernels and weights parameters are set to use a single fixed-point data type. In the first step, we implemented Lenet-5 using C/C++ and conducted simulation experiments. Once the experimental results met out requirements, the synthesis process converted the C/C++ code to HDL and the RTL model automatically. The synthesis report generated by the HLS gives a detailed information about the utilization of the different resources used in the project, such as Look-up table (LUT), Flip-Flop (FF), DSP Block, and Block Ram (BRAM), and the latency and the number of clock cycle needed. Next, we don't need to write a Verilog testbench code to conduct further verification, as the next C/RTL co-simulation step can do this for us. Furthermore, different FPGA on-chip environments are provided by Vivado HLS to simulate.

3 Optimization Method

HLS does not automatically optimize. In order to increase throughput and reduce the startup interval between operations, we need to choose different optimization strategies according to the characteristics of the task.

3.1 Function and Loop Pipelining

Pipelining allows operations to happen concurrently: each execution step does not have to complete all operations before it begins the next operations. Figure 5 shows the difference between functions with and without pipelining, and Fig. 6 shows the difference between for-loops with and without pipelining.

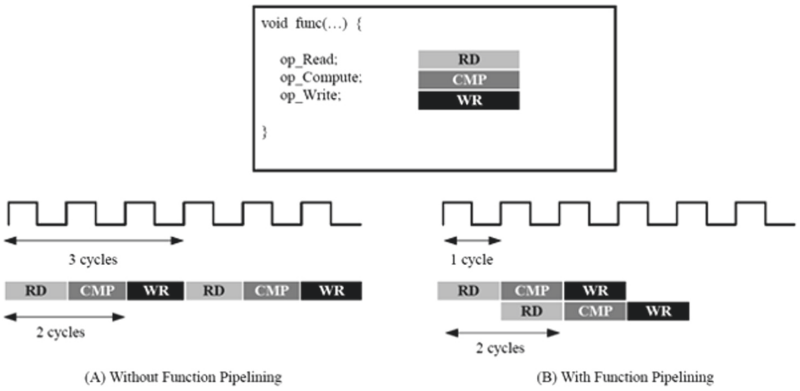

Fig. 5. Difference between functions with and without pipelining

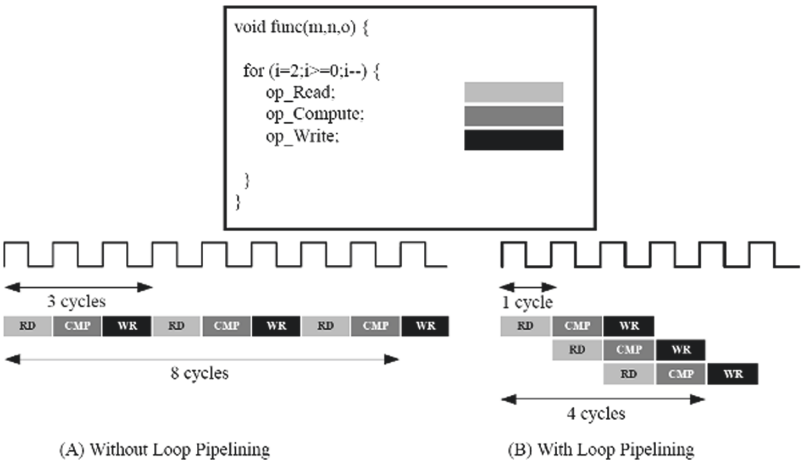

Fig. 6. Difference between for-loops with and without pipelining

3.2 Arrays Partitioning

Arrays are implemented as block RAM which only has a maximum of two data ports. This can limit the throughput of a read/write operations. The bandwidth can be improved by splitting the array into multiple small arrays, effectively increasing the number of ports. Figure 7 shows that array partitioning can be applied in 3 different manners. Figure 8 shows that partitioning dimension can be set when splitting a multi-dimension array.

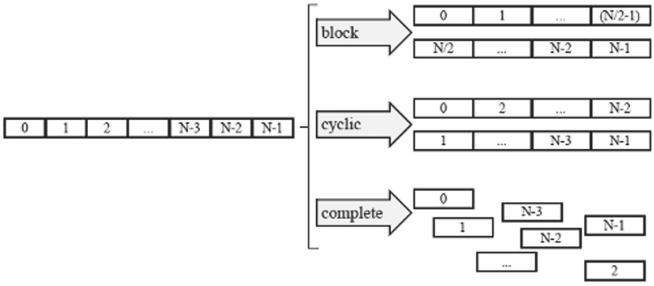

Fig. 7. Method for array partitioning

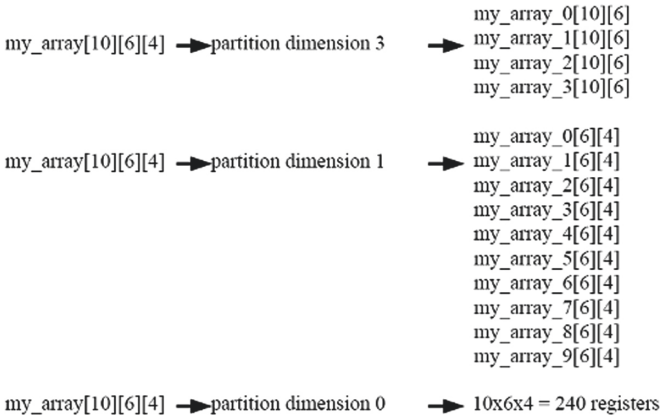

Fig. 8. Multi-dimension arrays partitioning

3.3 Loop Unrolling

In Vivado HLS, loops are kept rolled by default. These rolled loops generate a hardware resource which is used by each itertion of the loop. This is resource-efficient, but it can become a performace bottleneck. Using unroll directives, for-loops can be partially or completely unrolled to improve pipelining. Figure 9 shows how the loop unrolling actually works.

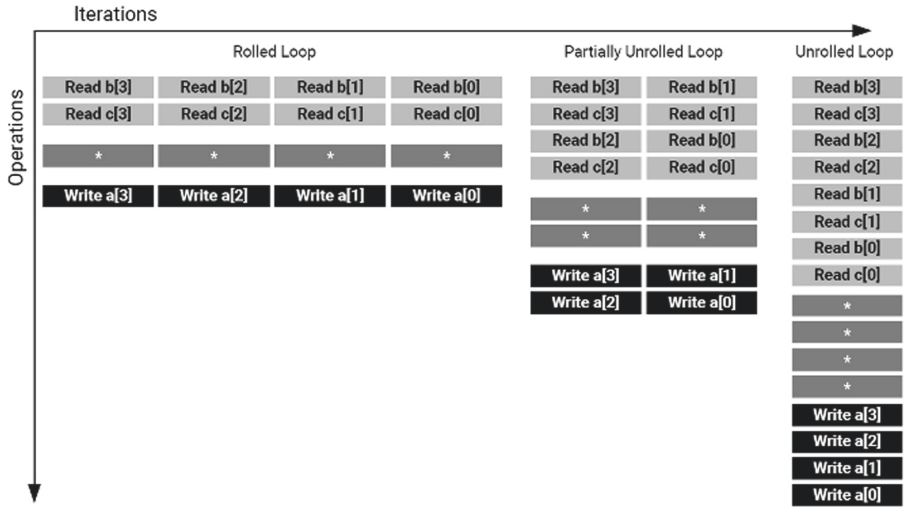

Fig. 9. How the loop unfolds

4 Implementing the Lenet-5 Using HLS

In this section, we discuss the acceleration methods for different modules in the neural network and compare the results.

4.1 Data Quantization

One of the most commonly used method for model compression is the quantification of weights and activations. In common development frameworks, neural network activations and weights are usually represented by floating-point data. Recent work attempts to replace this representation with low-level fixed-point data or even a small portion of training values. On the one hand, using fewer bits for each activation or weight helps reduce the bandwidth and storage requirements of the neural network processing system. On the other hand, using a simplified representation can reduce the hardware cost of each operation. Empirical results [4] also show that when choosing an appropriate method for data quantization, the impact on the accuracy of the model is acceptable. In [8], the optimized solution of a network is chosen layer by layer to avoid an exponential design space exploration. Guo et al. [3] choose to fine-tune the model after the fractional bit-width of all the layers are fixed. In our method, we use linear quantization method to optimize the model.

The selection of data quantization bit width has a great influence on the performance of the model. If the quantization bit width is too short, the weights that cause the model to be close to zero will be determined to be zero, which will cause many effective neurons to "dead", Affecting the performance of the model. If the quantization bit width is large, it will cause the model to consume a lot of resources and have a large latency.

Therefore, we selected the best quantization bit width through experiments. Table 1 shows the accuracy of the model under different quantization bit widths. When the quantization bit width is 9, the model cannot be synthesized due to the limitation of hardware resources. When the number of quantization digits is 7, the accuracy of the model will become very low. So in the end we used 8-bit quantization.

Table 1. Accuracy of the model under different quantization bit widths

Quantization width	7	**8**	9
Accuracy	0.129933	**0.972**	Cannot be synthesized

4.2 Optimization of Convolutional Layer

Figure 10 illustrates how a typical convolutional layer is computed [12].

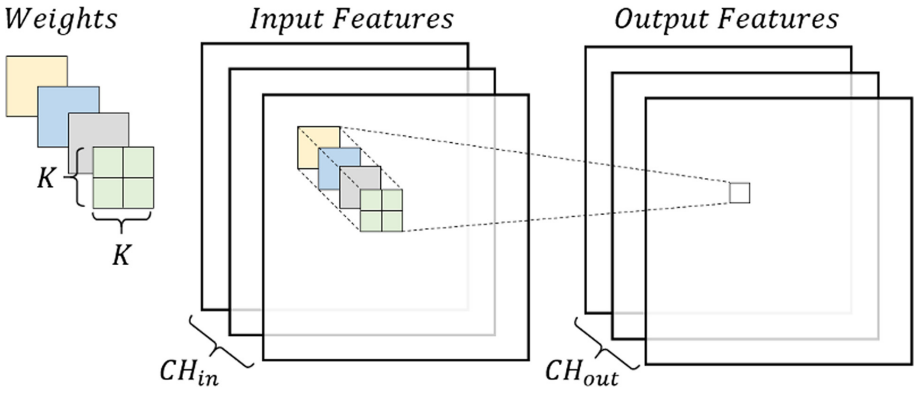

Fig. 10. How a convolutional layer is computed

The convolutional layer receives N input feature maps of size ROW_IN by COL_IN, and gives $N - K + 1$ feature maps of size ROW by COL as output. Each input feature map is convolved by a K by K kernel. The kernel slides through the corresponding input feature map and generates one pixel in the corresponding position in one output feature map. For simplicity, the stride of the kernel window here is set to one. Therefore, the size of each output feature map will be $N - K + 1$. Implementing the computing process of the convolution is basically just nesting 6 for-loops and do the multiplication and accumulation in the innermost loop. The code of a convolutional layer can be written in Algorithm 1.

This algorithm can be synthesized in to RTL model and HDL. Table 2 shows the parameters of this network layer. All the loops are kept rolled and no pipeline directives included. The total latency of this algorithm, latency of each nested loop, and the resources consumed is shown in Fig. 11. This program is resource-efficient, these rolled loops generate a hardware resource which is used by each iteration of the loop. But the total latency of this function is just too high.

Table 2. Parameters of this network layer

IN_CH	OUT_CH	ROW_IN	COL_IN	ROWS	COLS	K
6	16	14	14	10	10	5

Optimization techniques can be applied if we get rid of the data dependencies in this function. In the convolutional layer computing process described above, different input feature maps and output feature maps are independent can be computed in parallel. Pipeline directive can be applied if we just move the output channel loop and the input channel loop into the innermost loop, and partition the array in corresponding dimension. The nested loop inside the

Algorithm 1. Algorithm 1

```
1: Output_Channel:
2: for each cho ∈ [0, OUT_CH] do
3:    Input_Channel:
4:    for each chi ∈ [0, IN_CH] do
5:       ROWS:
6:       for each r ∈ [0, ROW] do
7:          COLS:
8:          for each c ∈ [0, COL] do
9:             Kernel_Row:
10:            for each kr ∈ [0, K] do
11:               Kernel_Col:
12:               for each kc ∈ [0, K] do
13:                  Out[cho][r][c]+ = In[chi][r + kr][c + kc] ∗ W[cho][chi][kr][kc]
14:               end for
15:            end for
16:         end for
17:      end for
18:   end for
19: end for
```

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Output_Channel	2037344	2037344	127334	-	-	16	no
+ Input_Channel	127332	127332	21222	-	-	6	no
++ Row	21220	21220	2122	-	-	10	no
+++ Col	2120	2120	212	-	-	10	no
++++ Kernel_Row	210	210	42	-	-	5	no
+++++ Kernel_Col	40	40	8	-	-	5	no

Fig. 11. Latency of Algorithm 1

pipeline directive will be automatically unrolled. The revised version of the algorithm is presented in Algorithm 2, and from the synthesis report shown in Fig. 12 we can see that the total latency has been shorten greatly, but it also consumes more resources.

By simply changing the order of the for-loops and applying several directives, the revised version of the algorithm is 102 times faster than the previous one. But still, further optimization method can be used to achieve an initial interval of one. If we notice the warning issued by the HLS complier, some data dependencies stop us from achieving a higher degree of parallelism. Figure 13 shows the RAW dependencies that exists in the nested for-loops. When executing iteration $0 (r = c = kr = kc = 0)$, Out[0][0][0] is read and then written into. When executing iteration $1 (r = c = kr = 0, kc = 1)$, Out[0][0][0] still needs to be read and written to. The for-loops cannot be fully pipelined if adjacent iteration needs to access the same piece of data.

Algorithm 2. Algorithm 2

```
1: ROWS:
2: for each r ∈ [0, ROW] do
3:    COLS:
4:    for each c ∈ [0, COL] do
5:       Kernel_Row:
6:       for each kr ∈ [0, K] do
7:          Kernel_Col:
8:          for each kc ∈ [0, K] do
9:             Output_Channel:
10:            for each cho ∈ [0, OUT_CH] do
11:               Input_Channel:
12:               for each chi ∈ [0, IN_CH] do
13:                  Out[cho][r][c] += In[chi][r + kr][c + kc] * W[cho][chi][kr][kc]
14:               end for
15:            end for
16:         end for
17:      end for
18:   end for
19: end for
```

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_Kernel_Row_Kernel_Col	20000	20000	9	8	1	2500	yes

Fig. 12. Latency of Algorithm 2

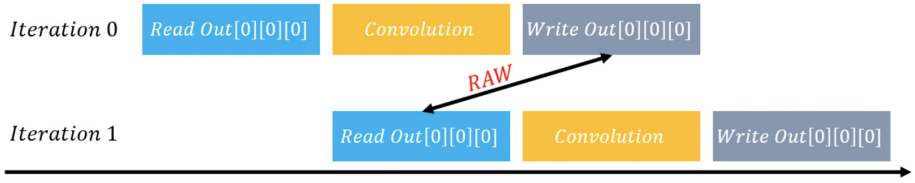

Fig. 13. RAW dependencies that exists in the nested for-loops

Simply changing the order of the for-loops by moving the $Kernel\_Row$ loop and the $Kernel\_Col$ loop to the outermost loop can remove these dependencies, shown in Fig. 14. The final version of the convolution algorithm is shown in Algorithm 3.

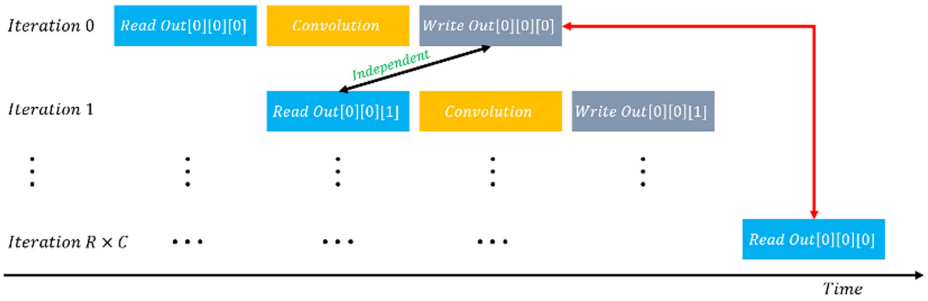

Fig. 14. RAW dependencies disappeared.

From the synthesis report shown in Fig. 15, initial interval of 1 is achieved, and the total latency is shortened to 25us, which is 812 times faster than Algorithm 1.

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Kernel_Row_Row_Col	2507	2507	9	1	1	2500	yes

Fig. 15. Initial interval of 1 is achieved.

Algorithm 3. Algorithm 3

```
1: Kernel_Row:
2: for each kr ∈ [0, K] do
3:    Kernel_Col:
4:    for each kc ∈ [0, K] do
5:       ROWS:
6:       for each r ∈ [0, ROW] do
7:          COLS:
8:          for each c ∈ [0, COL] do
9:             Output_Channel:
10:            for each cho ∈ [0, OUT_CH] do
11:               Input_Channel:
12:               for each chi ∈ [0, IN_CH] do
13:                  Out[cho][r][c]+ = In[chi][r + kr][c + kc] ∗ W[cho][chi][kr][kc]
14:               end for
15:            end for
16:         end for
17:      end for
18:   end for
19: end for
```

5 Experiment

5.1 MNIST

The MNIST dataset [1] (Mixed National Institute of Standards and Technology database) is a large database of handwritten digits collected by the National Institute of Standards and Technology. It contains a training set of 60,000 examples and a test set of 10,000 examples.

5.2 Implementation Details

During training, Our model uses the same structure as [10]. All of the input images are resized to 28×28. Moreover, We use the mini-batch of 8 and stochastic gradient descent (SGD) [7] for network training, the weight decay of 5×10^{-5}, and the momentum of 0.9. The learning rate is set to 0.001 and the iterations are set to 100 epochs.

During evaluation, we use the batch of 1. For each test, we first read the test picture from the disk through the Processing System (PS), and pass the image to the network acceleration module through direct memory access(DMA). In the network acceleration module, the digit in the image will be recognized, and the recognition result will be passed back to the PS through DMA and displayed.

5.3 Result

Table 3 shows the comparison between our FPGA-accelerated neural network and the same model running on PS. In the case of 8-bit quantization, our model loses less than 1% accuracy under the same hardware conditions, but the speed is accelerated by 3.64 times.

Table 3. Model comparison

	Accuracy	FPS
Processing System	0.982	71.43
FPGA-accelerated	0.976	19.62

6 Conclusion

In this paper, we have implemented FPGA-based handwritten digit recognition acceleration. By using data quantization and arrange the loops in an appropriate order, high-speed and low resources consumption are achieved.

References

1. Mnist (1998). http://yann.lecun.com/exdb/mnist/
2. Fpga design with vivado high-level synthesis ug998 (2019)
3. Guo, K., Sui, L., Qiu, J., Yu, J., Wang, J., Yao, S., Han, S., Wang, Y., Yang, H.: Angel-eye: a complete design flow for mapping CNN onto embedded FPGA. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(1), 35–47 (2018)
4. Guo, K., Zeng, S., Yu, J., Wang, Y., Yang, H.: A survey of FPGA-based neural network accelerator (2017)
5. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition (2015)
6. Huang, G., Liu, Z., van der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks (2016)
7. Lecun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
8. Qiu, J., et al.: Going deeper with embedded FPGA platform for convolutional neural network. In: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2016, pp. 26–35. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2847263.2847265
9. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition (2014)
10. Szegedy, C., et al.: Going deeper with convolutions (2014)
11. Xu, M., et al.: Stylize aesthetic QR code (2018)
12. Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., Cong, J.: Optimizing FPGA-based accelerator design for deep convolutional neural networks. In: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2015, pp. 161–170. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2684746.2689060