Automated Python-to-RTL Transformation and Optimization for Neural Network Acceleration

Chen Yang*, Renjing Hou†, Qirui Yang‡, Wenjian Yu§, Kang Zhao†
*School of Electronic Engineering, Beijing University of Posts and Telecommunications, Beijing, China
†School of Integrated Circuits, Beijing University of Posts and Telecommunications, Beijing, China
‡Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China
§Department of Computer Science and Technology, Tsinghua University, Beijing, China
Email: {yangch,hourj}@bupt.edu.cn, qyangau@connect.ust.hk, yu-wj@tsinghua.edu.cn, zhaokang@bupt.edu.cn

2024 2nd International Symposium of Electronics Design Automation (ISEDA) | 979-8-3503-5203-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISEDA62518.2024.10617696

***Abstract*—In order to optimize the process of accelerating large-scale neural network (NN) on field-programmable gate array (FPGA), this paper presents and optimizes the automatic flow based on HeteroCL and Xilinx Vitis HLS. This flow could transform python-based NN description to Verilog RTL running on FPGA. To improve its quality of result (QoR), many key optimization methods are proposed for the high-level synthesis (HLS) input, including fixed-point quantization, loop pipelining, convolutional buffer and others. To prove the feasibility of proposed optimization techniques, the convolutional NN (CNN) is selected as the experimental case study. And the results show that the delay and power consumption due to optimization techniques are significantly reduced.**

***Index Terms*—High-Level Synthesis, FPGA, Python-to-C, Compiling Optimization, Acceleration**

I. Introduction

Neural Network (NN) has been widely used in computer vision, natural language processing and other fields because of its excellent feature extraction ability. Field-Programmable Gate Array (FPGA) has the characteristics of low power consumption, low latency, parallel computing, and reconfigurability, which just meet the needs of neural network inference in machine learning, making it a high-quality carrier of neural networks.

With the increase of system technology and complexity, the high-level synthesis (HLS) technique is used to speed up the hardware acceleration based on FPGA. HLS has the ability to transform the logic structure described by high-level programming languages into the hardware implementation described by low abstract level languages. Vitis HLS from Xilinx is a successful commercial HLS tool, which has the advantages of improving development efficiency, accelerating design iteration, performance optimization, platform portability, and verification and debugging support. At the same time, its tight integration with other Xilinx tools and libraries makes FPGA development easier and more efficient.

However, although NN has a wide development and application prospect on FPGA, most NNs are generated by Python language based on frameworks such as Pytorch and TensorFlow, and few of them are implemented by C++ language that can support Vitis HLS. At the same time, Vitis HLS has limited optimization in the inference application of NN, and has high delay and power consumption, which cannot meet the application requirements. In order to solve these problems, our idea is to enhance an open-source toolchain to convert the NN described in Python into C++ code, and then convert to RTL code by Vitis HLS. Additionally, we customize several necessary LLVM (Low Level Virtual Machine) passes, leveraging the open-source front-end portion of the Vitis HLS [1], to optimize its application functionality. In the subsequent sections, we will choose CNN as a case study to illustrate our research and experimentally validate the feasibility of the optimization techniques.

This work is partially supported by National Key R&D Program of China (2022YFB2901100), and NSFC (No. 62090025).
Corresponding Author: Kang Zhao.

II. Related Work

In the previous work, Cornell University proposed HeteroCL [2] in 2019, an open-source tool that can be used to translate Python code into C++ code accepted by Vitis HLS. HeteroCL has a writing habit close to TensorFlow, and the extended Halide IR(intermediate representation) is converted into the code or executable files for each platform by the Halide compiler. It is very suitable for the deployment of CNN on FPGA. However, HeteroCL still has its shortcomings in the support of deep learning. After CNN is transformed into C++ code, there are some problems such as too many intermediate variables in the network, leading to memory shortage, simulation stack explosion, high delay and high power consumption.

Focusing on the problem of the network optimization, Jie Gong designed a general dynamic fixed-point quantization method for CNN [3]. The appropriate integer bits are found through formula (2.1), where x is the number to be quantized, l_{int} is the length of integer bits and $lbmax(x)$ represents the position of the highest bit (the leftmost non-zero bit) in the binary representation of x.

$$l_{int} = lbmax(x) + 1 \tag{2.1}$$

However, the output of each layer of the CNN also depends on the weight of the input of the layer. Adding one bit in this

Authorized licensed use limited to: Tsinghua University. Downloaded on August 13,2024 at 00:27:09 UTC from IEEE Xplore. Restrictions apply.

method cannot solve the overflow problem, and the workload is also increased by repeatedly adjusting the test.

Considering the influence of the input of each layer on the calculation of quantization bit width, Xiaokang Lei proposed Kullback-Leibler (KL) divergence to calculate the scale of the input parameter fixed-point [4]. This method reduces the amount of calculation while ensuring little loss of accuracy, but it requires a significant amount of pre-training to find the threshold, which requires large computational overhead for large-scale networks.

In terms of the optimization of convolutional calculation, Chen Zhang discussed the method of parallel convolution calculation of CNN in the HLS stage from two aspects of computational optimization and memory access optimization [5]. For loop unrolling, this paper divides whether the iterator corresponding to each loop layer participates in array addressing into three relationships. Different relationships produce different hardware implementations. and the loop corresponding to the iterator with the minimum cost is selected to unroll according to the relationship, which partially solves the problem of how to select loops for loop unrolling. However, the loop number and loop unrolling factor, which mainly affect the parallelism and computational cost of the network, are not discussed.

In order to solve some of the above problems and optimize the network performance as much as possible, the automation framework for FPGA agile development and the CNN inference optimization scheme based on FPGA designed by us are introduced in the following section, and the experimental evaluation results are given in the Results section.

III. Methods

In this section, we implemented an automated design flow. Based on the networks designed by the flow, we implemented optimization schemes such as quantized bit-width, data transmission through queues, memory access optimization through buffer optimization, and convolutional layer computation optimization.

A. Automated design flow for an agile FPGA development

In this automatic design flow, we hope that the final effect can be achieved:The user realizes a neural network using PyTorch and rewrote it to be implemented based on HeteroCL, and the subsequent steps of compiling the NN into the HLS code, synthesis, simulation and implementation are realized by the flow. Therefore, based on HeteroCL, we improved the process of designing neural networks on FPGA. The design and simulation verification process is illustrated in Fig. 1. The green part is completed by the programmer, while the blue part is automatically executed by the flow.

In the process, to facilitate users in building network models, we define the interface support for HeteroCL's deep learning, and the current supported interfaces are shown in TABLE I. By following the interface description and calling the interface functions, we can quickly build the required convolutional layers. For network layers that require extension, one only needs to invoke HeteroCL's interfaces and functions, such as *hcl.compute*, *hcl.reduce_axis*, *hcl.sum*, etc., for defining. This allows for the rapid declaration, iteration, and computation of tensor operators within the network.

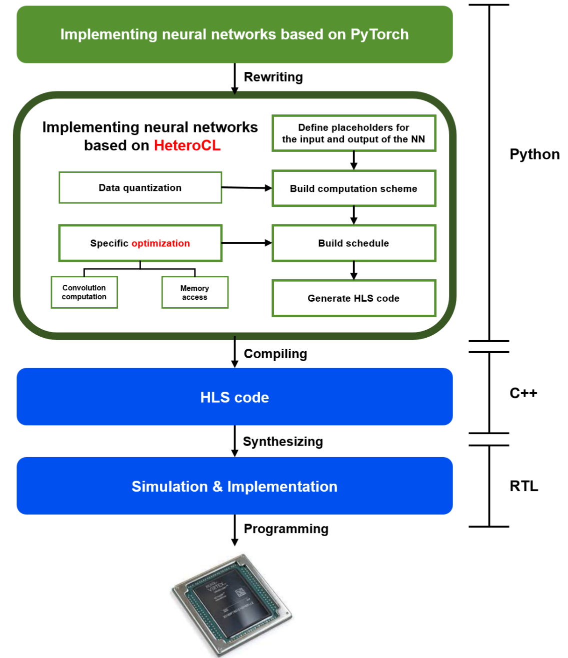


Fig. 1: Automated design flowcharts

Interface name	Interface description
conv2d_nchw	2D convolution layer without bias in nchw order
conv2d_nchw_bias	2D convolution layer with bias in nchw order
conv2d_nhwc	2D convolution layer without bias in nhwc order
avg_pool2d_nchw	2D average pooling layer in nchw order
avg_pool2d_nhwc	2D average pooling layer in nhwc order
maxpool2d	2D max pooling layer in nchw order
softmax2d	2D softmax layer in nchw order
logsoftmax2d	2D log softmax layer in nchw order
linear	2D linear layer in nchw order
relu	2D ReLU layer
flatten	Convert multi-dimensional array to 2D array
flatten_nchw	Convert 4D array in nchw order to 2D array
dense	2D fully connected layer

TABLE I: Interface name and description

After constructing the NN, in the process, the first step is to use the *hcl.placeholder* function to create a placeholder tensor for each input and output based on their dimensions. Then, the *hcl.create_scheme* function is called to build a Scheme, and quantization operations are performed at this stage. Next, based on the computation scheme, *hcl.create_schedule_from_scheme* is used to construct a Schedule. On the schedule, each Stage can be obtained using the name parameter, and interfaces can be invoked to optimize these stages specifically. Subsequently, the *hcl.build* function is called, and the parameters for the target are set. For the FPGA platform, the target is set to vhls, and the return value of *hcl.build* is the HLS code represented as a string. At this point, saving it to a file completes the process.

Authorized licensed use limited to: Tsinghua University. Downloaded on August 13,2024 at 00:27:09 UTC from IEEE Xplore. Restrictions apply.

In order to solve the high latency and high power consumption of CNN inference, We also define optimization options that can be enabled on Schedule, and the implementation ideas of specific functions will be given in the following subsections.

B. Fixed-point quantization scheme based on data distribution

This scheme is applicable to neural networks and is divided into global fixed-point quantization and local fixed-point quantization. Global fixed-point quantization is to find a bit width that can represent all the global numbers, and apply all the numbers involved in the calculation to this bit width. Local fixed-point quantization, on the other hand, computes a single bit width for each layer of the network and applies this bit width to all the numbers participating in the computation of that layer. Because local fixed-point quantization is equivalent to applying global fixed-point quantization to each layer, and HeteroCL automatically converts the bit-width when transferring data between layers with different bit-width, this section mainly introduces global fixed-point quantization.

Taking CNN as an example for illustration:For most CNNs with reasonable initialization and good training, their weights basically follow a certain statistical distribution. The distribution of the weights can be described by the maximum, minimum, average, median and variance of the statistical weights. Due to the data in the distribution set, the difference between the maximum value and the minimum value is small, the mean and median are close, and the variance is small, so these parameters can be used as the weight representative to determine the bit width. We use the maximum value, the minimum value and the average value to calculate the integer bit width and decimal place width respectively, and take the combination of the largest integer bit width and decimal place width as the bit width. This process is organized into the pseudo-code as shown in Algorithm 1.

Algorithm 1 Global fixed-point Quantization when the distribution is concentrated

Require $:max, min, mean$
1: Initialize maximum integer bit width max_int_width=0, maximum decimal bit width max_dec_width=0, fixed-point quantization bit width $width$
2: **for** $value$ in $[max, min, mean]$ **do**
3: Calculate the integer part of the value int_part and the decimal part dec_part
4: Integer bit width int_width=$[log_2(abs(int_part))]$+1
5: **if** $10 \le \frac{int_part}{dec_part} < 100$ **then**
6: dec_width=4
7: **else if** $\frac{int_part}{dec_part} \ge 100$ **then**
8: dec_width=1 or dec_width=0
9: **else if** int_part=0 or $\frac{int_part}{dec_part} \le 10$ **then**
10: dec_width=7 or $dec_width = 10, int_width$=0
11: **end if**
12: $max_int_width = max(max_int_width, int_width)$
13: $max_dec_width = max(max_dec_width, dec_width)$
14: **end for**
15: $width$=max_int_width+max_dec_width

In Algorithm 1, different integer and fractional bit widths are set based on the distribution of weights. When the value of the integer part of the weight is over a hundred times greater than the value of the fractional part, we decrease the precision of the fractional part by setting the fractional bit width to 1 or even 0. When the difference between the integer part and the fractional part of the weight is not significant, we increase the precision of the fractional part by setting the fractional bit width to 7 or 10 bits, ignoring the integer part.

In the algorithm, the bit width of the weight is determined, considering the influence of the input bit width on the output bit width, the output bit width can be calculated by equations (3.1) and (3.2). Here, O_{int} and O_{dec} represent the bit widths of the output's integer and decimal parts, I_{int} and I_{dec} represent the bit widths of the input's integer and decimal parts, and W_{int} and W_{dec} represent the bit widths of the weight's integer and decimal parts, respectively.

$$O_{int} = I_{int} + W_{int} + 1 \tag{3.1}$$

$$O_{dec} = max(I_{dec}, W_{dec}) \tag{3.2}$$

C. Data Transfer Method for Intermediate Layer

In the memory access optimization scheme for NN, we introduce the stream transmission structure, build a FIFO queue, and use the write and read interface to write and read data. On the premise of paying attention to the order of writing and reading data in different layers is consistent, the dataflow pragma in the main function (*top*) can be used to realize the parallel operation of the main function.

To realize the function of automatically converting inter-layer input and output into FIFO queues, we can call HeteroCL the *.to* method of the Schedule class, which is defined as follows:

def *to(self, tensor, dst=None, fifo_depth=-1)*

For the *dst* parameter, the target Stage to stream can be obtained by subscribing Schedule with name as described earlier, and similarly for the tensor parameter, *top.name* is used. Because the calculation of the network layer is pipelinated, the probability of blocking is small, usually the fifo_depth parameter is set slightly greater than the channel.

According to the aforementioned approach, employing FIFO queues for data read and write not only reduces data overhead but also facilitates computation pipeline between layers, thereby reducing network latency.

D. Convolutional buffer settings

In the previous section we set up the FIFO queue to reduce the network latency. But it should be noted that the convolutional access to an element tends not to be only once, so our reading after the FIFO queue introduces a row buffer and a window buffer. The buffer will record the input, while the convolution will be calculated by reading data from the buffer.

For the implementation of the buffer, we use the *reuse_at* function of Schedule, which is declared as follows, where

Authorized licensed use limited to: Tsinghua University. Downloaded on August 13,2024 at 00:27:09 UTC from IEEE Xplore. Restrictions apply.

target is the object to be reused, parent is the phase of reuse, and axis is the dimension in which data reuse occurs:

***def** reuse_at(self, target, parent, axis, name= None)*

The buffer is set so that when calculating the convolution, for large weight arrays, the weight value of a channel can be read directly through the window buffer, reducing the number of reads from external storage.

E. Parallel computation of convolution

In our design, we combine loop pipelining and loop unrolling to optimize the convolution computation for CNNs. According to the algorithm proposed in this paper [5], loop pipelining/loop unrolling is performed on the convolution selection dimension using FIFO queues and buffers. As shown in Algorithm 2, the main calculation parts *rc, ry* and *rx* correspond to the channel, height and width of the 3D convolution respectively, *xx* and *yy* correspond to the indices of the height and width of the output feature map. v131 corresponds to the index of the row buffer, and v135 and v136 respectively correspond to the vertical and horizontal indices of the window buffer. To achieve better results, we opted for loop pipelining/unrolling in the inner loop with no data dependencies and more computations, as illustrated by adding the pipeline pragma below the line for *for (int rc=0; rc<3; rc++)*. In HeteroCL, the same effect can be achieved by calling the *pipeline* method of Schedule and passing the corresponding dimension ($axis$) of the Stage corresponding to the loop as a parameter.

At the same time, in order to work with the loop pipelining and loop unrolling to produce the desired effect in Vitis HLS, we perform the array partitioning in the dimension of the loop unrolling/loop pipelining of the array participating in the calculation and hand it to the pipeline or parallel module for processing.

The above ideas are not only suitable for the optimization of parallel computation of convolution, but also for other computational layers.

F. Loop merging

Loop merging is also used to merge some network layers that are continuously calculated and the outer loop is exactly the same, without passing intermediate results, so as to greatly reduce the amount of memory access. This solution is only suitable for some networks. Such layers are usually BatchNorm layers and ReLU activation functions layers. These calculations do not change the input dimension and are usually performed continuously.

The implementation of this function relies on the Stage *.compute_at* function of HeteroCL, which stores the names of all layers in a list during the inspection of the network, and then slides through the list with a serial port of length two to check whether the names meet the prefix batchnorm and relu, respectively. If it is, the Stage is obtained from the Schedule with the name of the layer as the index. The Stage of the BatchNorm layer calls *compute_at* to aggregate the calculation of the BatchNorm layer into the loop dimension of the innermost layer of the ReLU layer to reduce the memory access and the number of network layers.

Algorithm 2 Example of 4D convolutions with FIFO queues and row and window buffering

```
1:  for (int nn=0;nn<2;nn++) do
2:     ...//Omitted outer loops
3:     for (int v131=0;v131<3;v131++) do
4:        //update the row buffer
5:     end for
6:     if (yy=2)>=0 then
7:        for (int v135=0;v135<3;v135++) do
8:           for (int v136=0;v136<3;v136++) do
9:              //update the window buffer
10:          end for
11:       end for
12:       if (xx-2)>=0 then
13:          float sum=0;
14:          for (int rc=0;rc<3;rc++) do
15:             for (int ry=0;ry<3;ry++) do
16:                for (int rx=0;rx<3;rx++) do
17:                   //convolution computation
18:                end for
19:             end for
20:          end for
21:          ap_fixed<10,4> v155=sum;
22:          conv1_x_0_conv1.write(v155);
23:          //HLS::Stream write
24:       end if
25:    end if
26: end for
```

IV. RESULTS

Based on the scheme before, this section takes three convolutional neural networks as examples to verify the rationality of the numerical bit width of the network determined by the fixed-point quantization scheme, and apply this bit width to verify the rationality and practical effect of the comprehensive optimization scheme.

A. Experimental Setup

The experiment was implemented on Xilinx Virtex7 with a clock cycle of 10ns. The software was based on Linux ubuntu 4.4.0-210-generic platform, Vitis HLS v2021.2 (64-bit), Vivado v2021.2 and HeteroCL v0.5 with MLIR.

B. Results of fixed-point quantization experiments

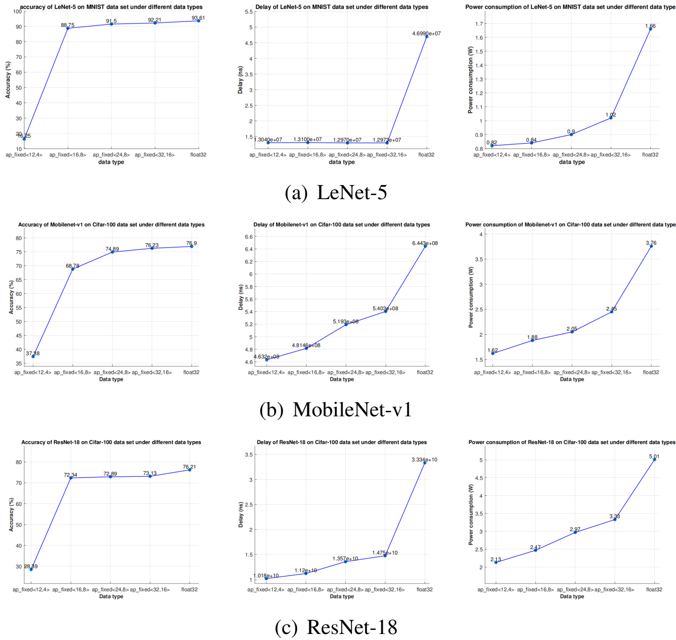
Network	Fixed point bit width	
	integer bit width	fractional bit width
Lenet-5	12	12
Mobilenet-v1	8	16
Resnet-18	8	16

TABLE II: Fixed-point quantization bit-width of each network calculated according to the fixed-point quantization scheme proposed in subsection B

Authorized licensed use limited to: Tsinghua University. Downloaded on August 13,2024 at 00:27:09 UTC from IEEE Xplore. Restrictions apply.

Network	Fixed point width		stream transfer+buffer		loop pipelining		loop unrolling		loop merging		Comprehensive optimization solutions		Baseline scenario	
	Integer bit width	fractional bit width	delay(ns)	power consumption (W)	delay(ns)	power consumption (W)	delay(ns)	power consumption (W)	delay(ns)	power consumption (W)	delay(ns)	power consumption (W)	delay(ns)	power consumption (W)
LeNet-5	12	12	2.11E+07	1.05	5.34E+06	0.98	4.97E+06	1.12			**5.55E+06**	**1.14**	4.70E+07	1.66
MobileNet-v1	8	16	5.71E+08	2.34	4.06E+07	2.08	3.84E+07	2.47	6.38E+08	1.89	**4.44E+07**	**2.12**	6.44E+08	3.76
ResNet-18	8	16	1.88E+09	3.63	8.43E+08	3.11	8.03E+08	3.58	1.33E+10	2.63	**8.86E+08**	**2.98**	3.33E+10	5.01

TABLE III: Experimental results of delay and power consumption for ablation experiments optimized separately for each scheme and for the integrated optimization scheme



(a) LeNet-5

(b) MobileNet-v1

(c) ResNet-18

Fig. 2: Line plots of accuracy, delay and power consumption on the respective data sets for the three baseline network architectures with four different fixed-point bit widths and 32-bit floats

In the experiments, LeNet-5 used MNIST dataset for handwritten digit classification task (10 classifications), MobileNet-v1 and ResNet-18 used Cifar-100 dataset for image classification task (100 classifications), and the input batch size was 2. The rationality of the scheme is quantified by the prediction accuracy of fixed-point numbers and floating-point numbers. Compared with the weights obtained by Pytorch training under the same conditions, the experimental results are LeNet-5:93.6%, MobileNet-v1:77.4%, and ResNet-18:76.2%, respectively, which are similar to the information on the public leaderboard [6], and the pre-training weights are reasonable.

TABLE II shows the results of fixed-point quantization which are derived based on the data quantization scheme mentioned in Section III-B. At the same time, three different fixed-point bit widths were selected in the experiment for comparison with floating-point numbers in terms of accuracy, delay, and power consumption. In order not to affect the efficiency of hardware implementation, the selected four fixed-point bit width is aligned to 4 bits.

The experimental results of data quantization schemes for the three networks are shown in Fig. 2. The final experimental results showed the quantization scheme has about 3% difference in accuracy with floating-point number, and the delay and power consumption are significantly lower than that of floating-point number. It proves that the fixed-point quantization scheme in this paper is reasonable and effective.

C. Inference optimization experimental results

According to the previous scheme, the final composition of the comprehensive optimization scheme of the three networks is shown in TABLE IV. In the table, we use ✓ for applicable, and ✗ for deprecation. At the same time, we designed ablation experiments to reflect the optimization effects. Through Vitis HLS, co-simulations were conducted separately for the three network susing the fixed-point bit width data in TABLE II, obtaining the resource expenditures before and after applying their respective optimization methods, as shown in TABLE III.

network	fixed point quantization	stream transfer+buffer	loop pipelining	loop unrolling	loop merging
Lenet-5	✓	✓	✗	✓	✗
Mobilenet-v1	✓	✓	✓	✗	✓
ResNet-18	✓	✓	✓	✗	✓

TABLE IV: Composition of comprehensive optimization schemes for the three networks

It can be seen from the table that under the action of the comprehensive scheme, the delay and power consumption of LeNet-5 are reduced to 11.8% and 63.7%, respectively. The delay and power consumption of MobileNet-v1 are reduced to 6.9% and 56.4%, respectively. The delay and power consumption of ResNet-18 are reduced to 2.66% and 59.5%, respectively, which reflects the effectiveness of the comprehensive optimization scheme.

V. Conclusions

This paper proposes and implements an FPGA-based optimization process for Neural Network (NN) inference. Through HeteroCL, Vitis HLS and Vivado tools, we successfully implemented a process that can automatically deploy a CNN implemented in Python to FPGA. In the HLS stage, the optimization methods including quantization, loop pipelining and loop merging are implemented, which reduces the delay and power consumption of network operation, and significantly improves the efficiency of network operation.

Authorized licensed use limited to: Tsinghua University. Downloaded on August 13,2024 at 00:27:09 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] Xilinx. Xilinx hls llvm project[EB/OL]. https://github.com/Xilinx/hls-llvm-project.

[2] Lai Y H, Chi Y, Hu Y, et al. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing[C]//Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2019: 242-251.

[3] J. Gong, S. Zhao, H. He, et al. "Design of Quantized CNN Acceleration System Based on FPGA"[J]. Computer Engineering, 2022(3):170-174.

[4] X. K. Lei, Z. G. Yin and R. L. Zhao. "FPGA-based convolutional neural network fixed-point acceleration"[J]. Journal of Computer Applications, 2020(10):2811-2816.

[5] C. Zhang, P. Li, G. Y. Sun, Y. J. Guan, B. J. Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks[C]//Proceedings of the 2015 ACM/SIGDA international symposium on field programmable gate arrays. 2015: 161-170.

[6] Code P W. Image classification on cifar-100 - state-of-the-art[Z]. 2023.

Authorized licensed use limited to: Tsinghua University. Downloaded on August 13,2024 at 00:27:09 UTC from IEEE Xplore. Restrictions apply.