Fully Pipelined FPGA Acceleration of Binary Convolutional Neural Networks with Neural Architecture Search

Ji, Mengfei; Al-Ars, Zaid; Chang, Yuchun; Zhang, Baolin

DOI
10.1142/S0218126624501706

Publication date
2024

Document Version
Final published version

Published in
Journal of Circuits, Systems and Computers

Journal of CIRCUITS, SYSTEMS, AND COMPUTERS

Volume 33 • Number 10 • 15 July 2024

World Scientific

NEW JERSEY • LONDON • SINGAPORE • BEIJING • SHANGHAI • HONG KONG • TAIPEI • CHENNAI • TOKYO

Journal of Circuits, Systems, and Computers
Vol. 33, No. 10 (2024) 2450170 (19 pages)
© World Scientific Publishing Company
DOI: 10.1142/S0218126624501706

Fully Pipelined FPGA Acceleration of Binary Convolutional Neural Networks with Neural Architecture Search*

Mengfei Ji†
College of Electronic Science & Engineering,
Jilin University, Jilin, China
The Department of Quantum & Computer Engineering,
Delft University of Technology, Delft, Netherlands
jimengfei007@outlook.com

Zaid Al-Ars
The Department of Quantum & Computer Engineering,
Delft University of Technology, Delft, Netherlands
z.al-ars@tudelft.nl

Yuchun Chang
School of Microelectronics,
Dalian University of Technology, Dalian, China
cyc@dlut.edu.cn

Baolin Zhang
College of Electronic Science & Engineering,
Jilin University, Jilin, China
zbl@jlu.edu.cn

Received 21 June 2022
Accepted 27 November 2023
Published 14 February 2024

In this paper, we present a fully pipelined and semi-parallel channel convolutional neural network hardware accelerator structure. This structure can trade off the compute time and the hardware utilization, allowing the accelerator to be layer pipelined without the need for fully parallelizing the input and output channels. A parallel strategy is applied to reduce the time gap in transferring the output results between different layers. The parallelism can be decided based on the hardware resources on the target FPGA. We use this structure to implement a binary ResNet18 based on the neural architecture search strategy, which can increase the accuracy of manually designed binary convolutional neural networks. Our optimized binary ResNet18 can achieve a Top-1 accuracy of 60.5%

*This paper was recommended by Regional Editor Emre Salman.
†Corresponding author.

on the ImageNet dataset. We deploy this ResNet18 hardware implementation on an Alphadata 9H7 FPGA, connected with an OpenCAPI interface, to demonstrate the hardware capabilities. Depending on the amount of parallelism used, the latency can range from 1.12 to 6.33 ms, with a corresponding throughput of 4.56 to 0.71 TOPS for different hardware utilization, with a 200 MHz clock frequency. Our best latency is 8× lower and our best throughput is 1.9× higher compared to the best previous works. The code for our implementation is open-source and publicly available on GitHub at https://github.com/MFJI/NASBRESNET.

Keywords: FPGA accelerator; OpenCAPI; high throughput; low latency; fully pipelined; flexible parallelism.

1. Introduction

Convolutional neural networks (CNNs) have many application scenarios in many computer vision tasks in image recognition, image classification and object detection because of their high accuracy.[1–3] FPGA accelerators enable CNNs to have better performance in terms of low latency, low power consumption and high bandwidth to be better applied to edge devices or other strict application scenarios. Binary convolutional neural networks (BNNs), which are convolutional neural networks with binary weights and activations,[4–6] are friendly to implement on FPGAs because they will use fewer hardware resources.[7] However, this results in two challenges: (1) low accuracy of BNNs and (2) complexity of hardware implementations.

With regard to the first challenge, the accuracy of BNNs is much lower than the accuracy of their full-precision CNNs counterparts. Existing solutions focus on software-based improvement of these models using multi-weight binary models[8] or linear combinations of binary weights.[9] However, these methods only address the low bit representation of the weights rather than improving the architecture of the CNN. Neural architecture search (NAS) strategy[10,11] is efficient in improving the accuracy of CNNs by optimizing the CNN architecture itself. Applying neural architecture search strategy to BNNs can improve their accuracy,[12] thereby making architecture search strategies suitable to implement on FPGAs because of their low hardware utilization and higher accuracy.

With regard to the second challenge, many CNN accelerators have appeared in recent years. One popular accelerator structure is the general CNN accelerator based on the system-on-chip structure.[13] These accelerators are suitable for a wide range of different CNNs, but they are not able to achieve the maximum performance possible for any specific neural network. Especially when the network is a multi-branch network such as ResNet[14] or other related networks,[15] the data transmission between the CPU and the hardware will take a large amount of time. Another well-known structure is the pipeline structure.[16] This structure is based on the parallelism between channels, which consumes a lot of hardware resources that could limit its practicality due to the limited FPGA resources available on chip. When doing an FPGA design, the latency and the hardware resources are both important to be considered.

In this paper, we present a pipelined CNN FPGA structure with flexible parallelism, which allows the layers to be fully pipelined without the need for fully parallel channels. The parallel strategy can deal with the gap in the transfer time of results between different layers, which allows the system not to need to wait for the results while operating to reduce latency. The pipelined structure makes the latency of the network low, and flexible parallelism makes it suitable for FPGAs with a reduced amount of hardware resources. We show an implementation of binary ResNet18 with a neural architecture search strategy to improve accuracy and quantizing methods to reduce hardware utilization using this structure. Our system uses the OpenCAPI interface to ensure high bandwidth communication with the host processor and prevent data communication bottlenecks.

The contributions of this paper are as follows:

(1) We show a hardware accelerator model able to trade off compute time and hardware utilization, which is layer-pipelined and has flexible-parallel channels.
(2) We further optimize the quantized binary ResNet18 network to reduce its hardware utilization.
(3) We integrate the accelerator with the high-bandwidth OpenCAPI interface and measure its performance.

The rest of the paper is organized as follows. In Sec. 2, we discuss related background research work of CNN accelerators. In Sec. 3, we introduce the NASB strategy and present the architecture of the CNN we implement. Section 4 introduces a novel hardware structure and the way we implement the network on FPGA. Section 5 provides an evaluation of the performance this design is able to achieve. Section 6 concludes the paper.

2. Related Work

A large number of CNN accelerators have been designed to have better performance using different technologies. Published work[17,18] uses a systolic array architecture to improve the efficiency of DSPs and save LUTs. Some researchers[19] take advantage of FPGA dynamic reconfigurability to design circuits fully automatically to get better performance. Other research[20,21] compresses the network using pruning to reduce the size of the model and have a better power efficiency. However, all these publications do not use model quantization methods but focus on other approaches to reduce model size. In contrast, instead of reducing model size, other published work[22] proposes using larger FPGAs in the cloud instead of local FPGAs for deploying CNNs because of their larger resource availability. However, this approach increases the inference latency significantly due to the delay in edge-to-cloud communication.

Binarization, as an important method for model quantization, is also widely used in the design of CNNs. A lot of FPGA designs with binary neural networks have appeared in recent years. YodaNN[23] uses binary weights to reduce the complexity of the computing unit and the bandwidth to transfer the weights to external memory. This means binary weights can reduce the complexity of the circuit, thereby reducing power consumption and area. FP-BNN[24] removes the bottleneck involving multipliers by bit-level XNOR and shifting operations, and the bottleneck of parameter access by data quantization and optimized on-chip storage. Other research[25] proposes a scalable fully pipelined BNN architecture, which targets maximizing throughput and reducing energy and resource efficiency in large FPGAs, in addition to proposing a methodology to explore design space to ensure optimal configuration. However, these efforts only focus on the performance of hardware implementation, they do not work on increasing the accuracy of the models.

The related work shows that the trade-off between accuracy, compute time and hardware utilization is still an open research topic that has not yet been addressed by literature. In this paper, we propose using hardware-optimized models using architecture search methods in combination with advanced binarization techniques to minimize model size.

3. NASB-CNN Model

In this section, we introduce the NASB strategy and we present the NASB-ResNet18 model architecture we implement in this paper.

3.1. *NASB strategy*

The NASB strategy is implemented to apply network architecture search to binary CNNs.[12] The NASB strategy consists of three main stages: the searching stage, pretraining stage and finetuning stage.

In the searching stage, we search for an optimized architecture of BNN on a small dataset. A NASB-convolutional cell is applied as an optimized architecture for binarizing its full precision counterpart. The connections of the NASB-convolutional cell in this stage are shown in Fig. 1. In Fig. 1(a), the left part is the backbone of

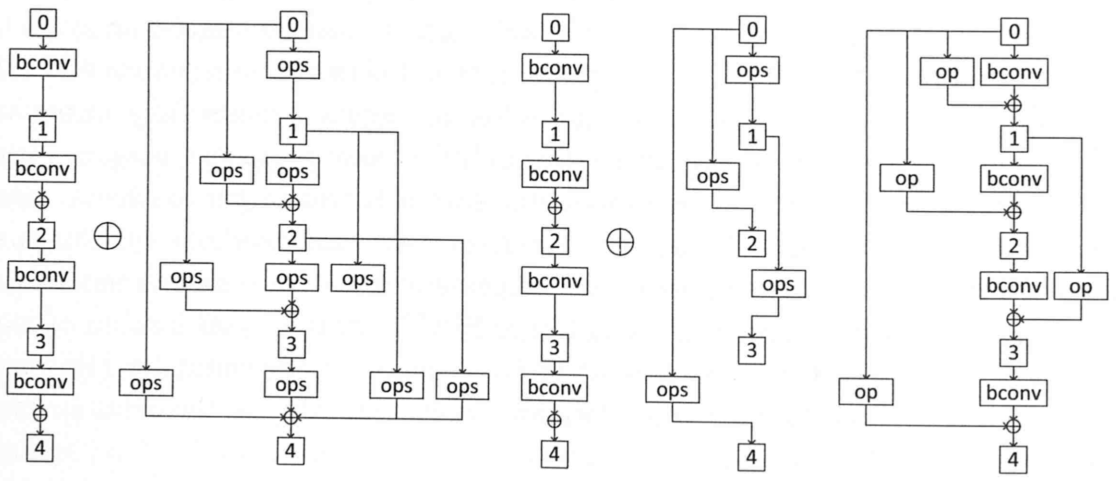


Fig. 1. Connections of the NASB-convolutional cell.

Table 1. Operations for the NASB–convolutional cell.

Operation	Description
op0	Zero
op1	3×3 average pooling
op2	3×3 max pooling
op3	Identity
op4	1×1 convolution
op5	3×3 convolution
op6	5×5 convolution
op7	1×1 dilated convolution
op8	3×3 dilated convolution
op9	5×5 dilated convolution

the NASB-convolutional cell, and the right part is a NAS-convolutional cell. The bconv in the figure represents a binary convolutional layer and the ops represents the operations that can be used for the architecture search. Figure 1(b) shows the backbone and the NAS-convolutional cell after being searched. Figure 1(c) is the final architecture of the whole NASB-convolutional cell using the NASB strategy. The operations for the NASB-convolutional cell are listed in Table 1.

In the pretraining stage, we first use the architecture that is defined in the searching stage to create a full-precision CNN model by replacing the bconv layers with full-precision layers. Then, we train the full-precision CNN model using the target dataset.

Finally, in the finetuning stage, we binarize the full precision CNN model to a BNN model and train it on the target dataset. More information about the details of the NASB strategy can be found in the literature.[12]

3.2. *Model architecture*

In this paper, we use the NASB strategy to search the architecture and train a binary ResNet18.[12] The final architecture of the NASB-ResNet18 model is shown in Fig. 2. The black-colored blocks are part of the original ResNet18, which is the backbone of the NASB-convolutional cell, while the gray-colored blocks are the branches we add to ResNet18 using the NASB strategy. The Bconv layer is a binary convolutional layer. We do Sign activation before every binary convolutional layer, and each binary convolutional layer is followed by a ReLU layer and a batch normalization layer. Every max-pooling layer is followed by a batch normalization layer.

As this network is a binary ResNet18, the weights in binary convolutional layers are 2-bit (bipolar: −1 or 1) and the activations are 1-bit. As shown in Fig. 2, we can see that the non-binary parameters include the weight and bias in the batch-normalization layers, the weight of the convolutional operations in downsample layers and the weight of the fully connected layers. We use the rounding method to quantize the decimal part of the non-binary parameters into 16 bits[9,26] and the input images into 8 bits. According to the experimental results and our experience,

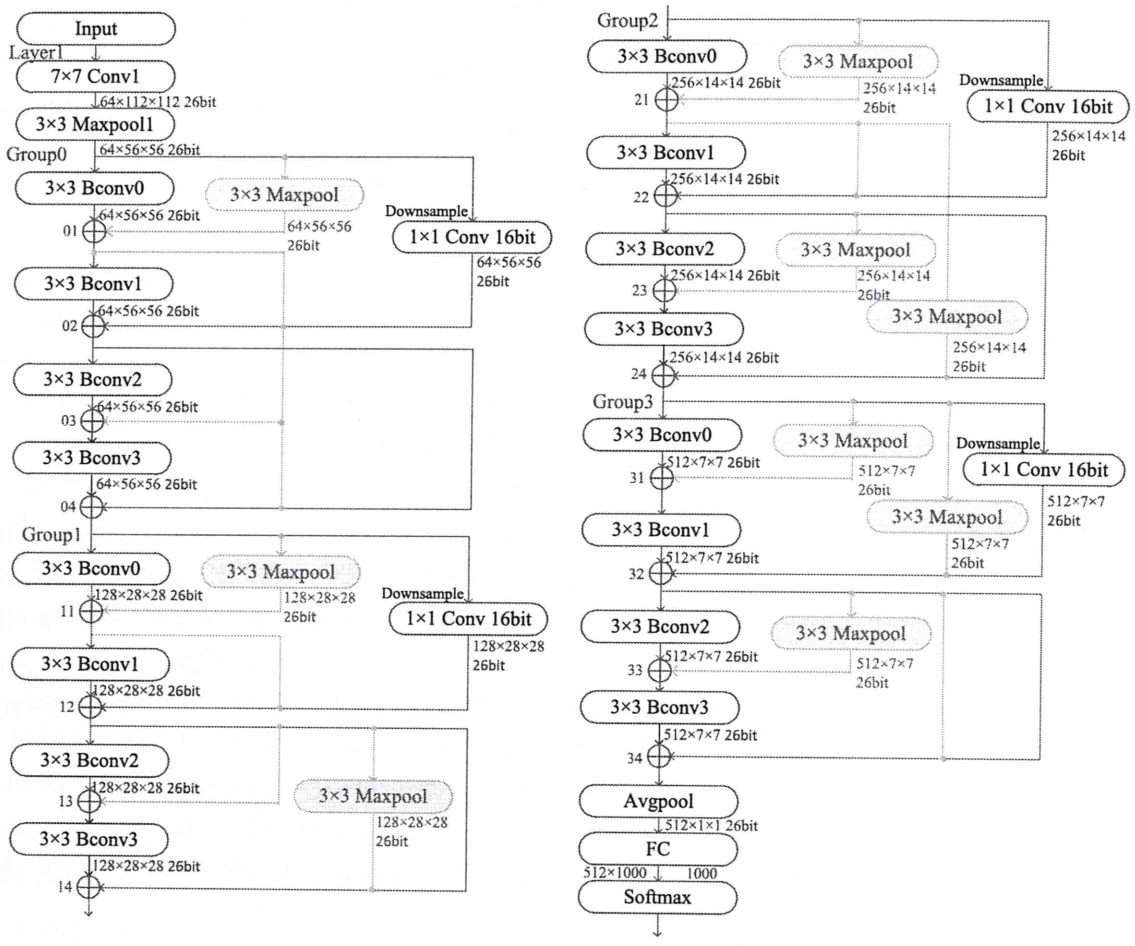


Fig. 2. The architecture of the NASB-ResNet18 model.

the bandwidth of each output feature can be set to 26 bits, and this does not lead to any accuracy loss. The Top-1 accuracy of the NASB-ResNet18 in this design on the ImageNet dataset is 60.5%.

4. Hardware Implementation

In this section, we present the detailed hardware structure of each component. We introduce a hardware structure that ensures a trade-off between the hardware utilization and the compute time.

4.1. *Overall hardware structure on OpenCAPI*

In this paper, we integrate our design using the OpenCAPI interface, as shown in Fig. 3. OpenCAPI provides an easy controlling platform with fast data transfer from the host server to the accelerator. We transfer the parameters and the images from the host side to the hardware side through a 512-bit AXI bus. A serializer and a decoder will send these parameters to different layers. Then these parameters will be stored on the chip to be used by the CNN following the order of kernel, input channel and output channel.

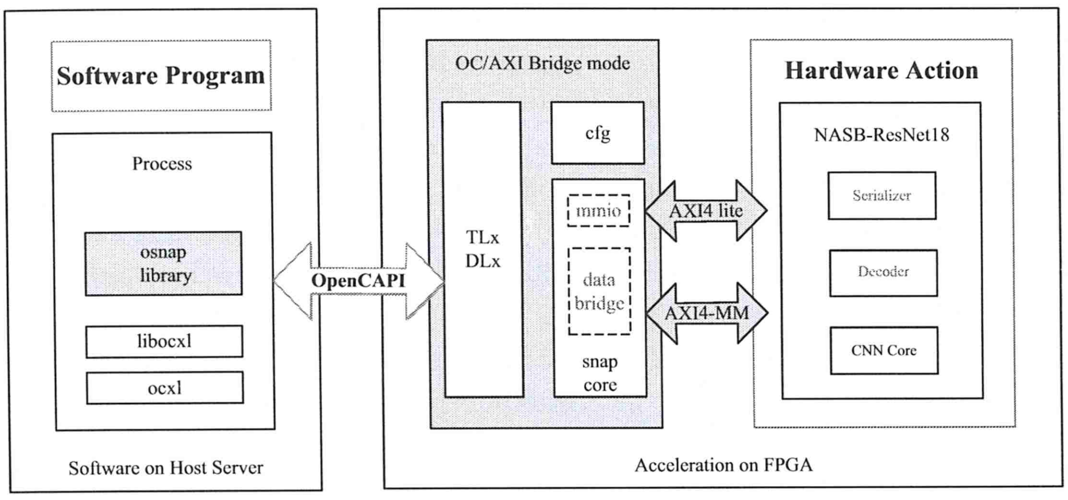


Fig. 3. OpenCAPI hardware architecture.

4.2. *Hardware structure of NASB-ResNet18*

To achieve high accuracy, CNNs are usually very deep and have a large number of parameters. In the binary CNN, not all layers are binarized. The main layers that are usually not binarized are the batch-normalization layers and downsample layers, among others. This way, the accuracy can be kept at a high enough level. However, this also results in high hardware utilization for the fully pipelined and fully parallel structure of the network, also due to the fact that these fully parallel structures contain many channels. To ensure a trade-off between the compute time and the hardware utilization, we propose a layer pipeline and semi-parallel channel hardware structure. In general, this represents a pipeline between layers, so this structure makes efficient use of FPGA resources and results in a compute time that is much shorter compared to CPUs. Apart from this, the amount of parallelism between output layers can be decided depending on the available hardware resources on the FPGA.

To better illustrate the trade-off this hardware structure ensures between hardware utilization and compute time, we discuss our data flow design of each layer. For all the layers in this network, the data flow between layers is based on the row-channel structure, which means the pixels are scanned in row-by-row for all different channels in that given row. Figure 4 presents the row-channel data structure. This data structure allows the hardware to start to operate on the next layer before the former layer fully finishes, although the operation of the output channels is not fully parallel.

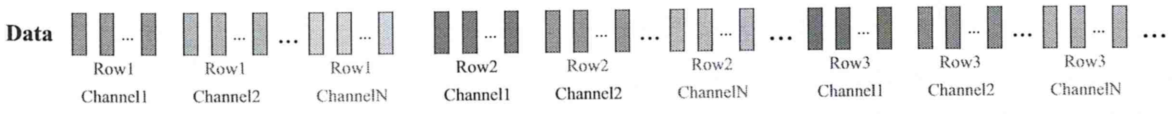


Fig. 4. Row-channel data structure.

Given this data structure, we can introduce the hardware structure that is able to trade off hardware utilization and compute time. The design of Layer1 shown in Fig. 2 can be regarded as the most important part of the network, which applies a parallel strategy for output channels to reduce the time gap of transferring the output results between different layers. Following the row-channel data structure, we can see that the time that the input image needs to transfer its data is $224 \times 224 \times 3$ clock cycles, since the image size is 224×224 with 3 channels. Furthermore, the time that the image need to transfer its data after the 7×7 Conv1 convolutional block of Layer1 is $112 \times 112 \times 64$ clock cycles. This means that there is a big increase in the transfer time. However, after the 3×3 Maxpool1 Layer, the transfer time decreases to $56 \times 56 \times 64$ clock cycles. The time to transfer the results of the Maxpool1 Layer is a quarter of the time to receive the results from the Conv1 Layer, which is also the time to operate the max-pooling operations in the Maxpool1 Layer. If we fully follow the row-channel data structure, there will be a lot of time wasted while transferring data. To solve this problem, we can use multiple threads to operate the 7×7 Conv1 Convolutional Layer concurrently. The remaining time after the Maxpool1 Layer of one thread can be used to transfer the results of the other threads. When we use two threads, each thread operates on 32 output channels, and still half of the transfer time for the results after the Maxpool1 Layer is vacant. When we use four threads, the time for the results after the Maxpool1 Layer fits the operation time well, so there is no time gap to transfer the output of the Maxpool1 Layer. When we use eight threads or more, not only is there no time gap, but also we need to use multiple wires to transfer the results to the next stage, which will also cause higher parallelism for the operations in the following groups. The timing diagram of the output after Maxpool1 in Layer1 of one thread to eight threads is shown in Fig. 5.

However, when we use multiple threads, the data flow becomes faster, and the hardware utilization will increase. We will analyze the hardware utilization difference while using different threads later in the paper. The number of threads to implement the network can be decided by the available resources on the target FPGA.

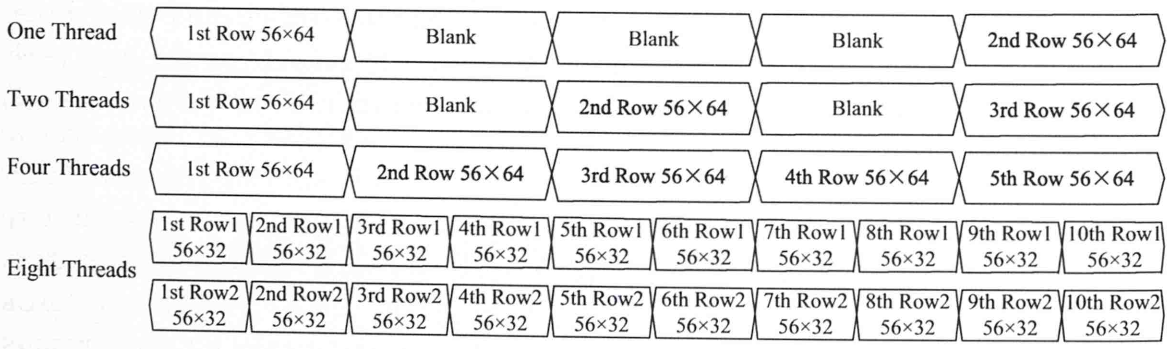


Fig. 5. The timing diagram of the output after Maxpool1 with different threads.

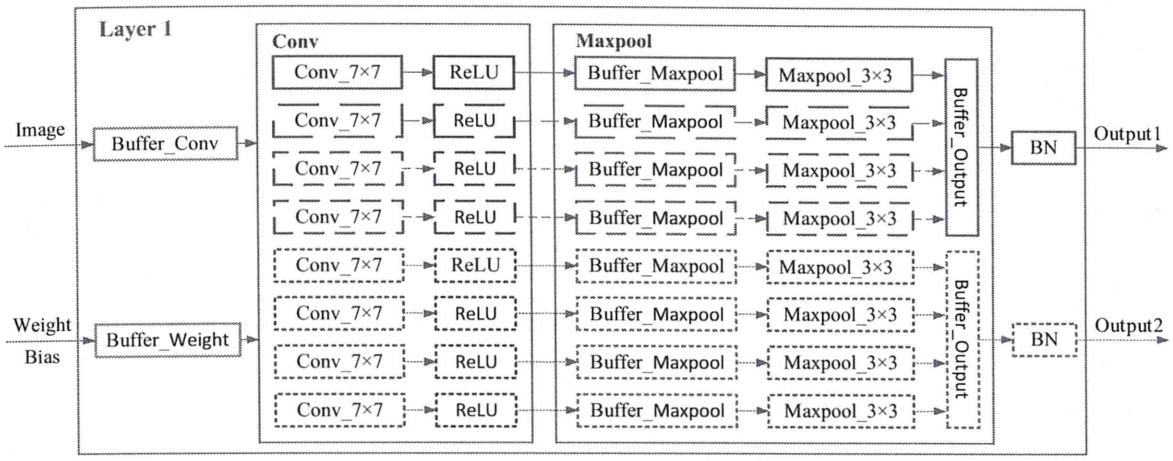


Fig. 6. Structure of Layer1.

When operating the Conv1 Layer, because this convolution is a 7×7 convolution, we use FIFOs to store the data of the first six rows, and when the seventh row flows in, the convolutional operation can start. The usage of the FIFOs for the following layers all follow the same rule. The results of the multiple threads are sent to the Maxpool1 Layer simultaneously. There are also multiple threads for the max-pooling layer to handle the data from the Conv1 Layer. Then we store the results of one row of the multiple threads in a data buffer and then transfer the results to the next stage using the row-channel data structure. The structure of Layer1 is shown in Fig. 6.

The data structure in Group0 in Fig. 2 also follows the row-channel data structure. The convolutional layers in Group0 follow the layer-pipelined and fully parallel output channel hardware structure. For 3×3 binary convolutional layers, we store the first two rows of data with all input channels in FIFOs after doing the Sign operation. We use this data to calculate the results of the first row of the first input feature map. After this, we store the first row of results in a RAM. Then we calculate the results of the same row of the second input feature map and add the results of the first feature map simultaneously. With the same strategy, when we finish calculating all input feature maps, we get the final results of the first row of the output feature map. All the output channels are fully parallel when doing the convolution operation, which means that we get the results of all output maps simultaneously. We store the results of one row for all output channels in a buffer and then send it to the next stage to do ReLU and batch-normalization following the row-channel data structure. All the multiplications in binary convolutional layers are based on Multiplexers instead of DSPs to save hardware resources. This convolution operation is shown in Fig. 7. For the Downsample Layers, the procedure is almost the same as the binary convolutional layers. However, the Downsample Layers use DSPs to do multiplication. For max-pooling layers, we also use FIFOs to store the first two rows, and we use this data to find the max and get the results one by one.

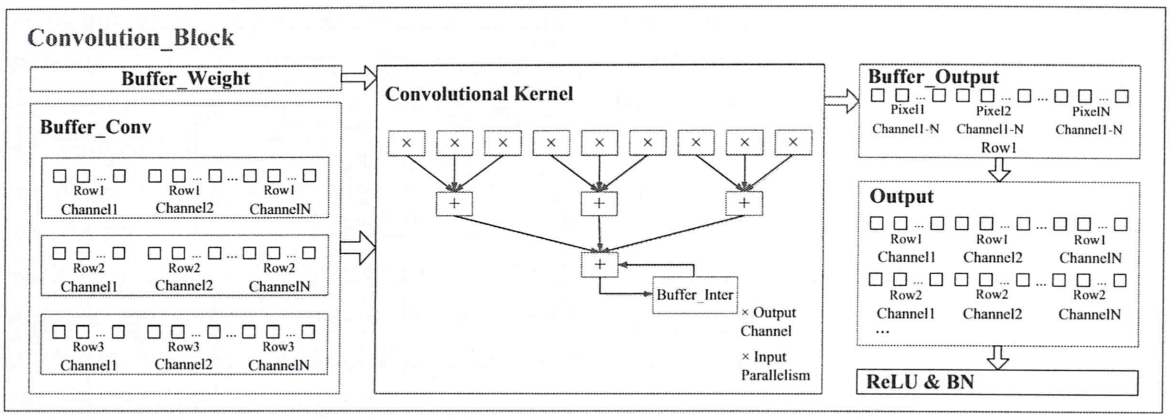


Fig. 7. Hardware structure of binary convolutional layers in groups.

From Fig. 2, we can see that there are multiple branches in this group that need to be added together. We use a pipeline structure and add these branches in a streamed pipelined fashion. We store the first several rows in FIFOs if they need to wait for other operations. We get them out from FIFOs when the latest operation of this node begins. We take Node 02 of Group0 for example. When the input data of Group0 comes, it goes through both the first 3×3 convolutional operation and the 3×3 max-pooling operation. At the same time, the first five rows of the input are stored in a FIFO prepared to do the 1×1 convolution and be added to Node 02. When we get the result of Node 01, we operate the 3×3 convolutional operation and store the first two lines of the result into another group of FIFOs. By the time the first result of the second convolutional layer comes out, the data input of this group is streamed to the sixth row, and the output of Node 01 is streamed to the third row. Then we get the data out from the first group of FIFOs and do the 1×1 convolution. At the same time, we get out the data from the second group of FIFOs and then add all these three branches together. With this technology, multiple branches of data can be added simultaneously. The structure of Group0 is shown in Fig. 8. The maxpool block shares the same input buffer with

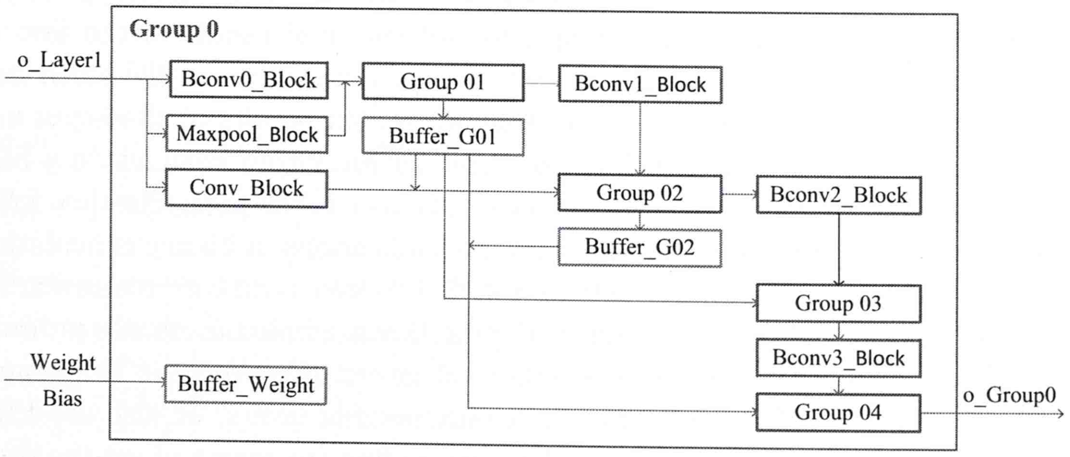


Fig. 8. Structure of Group0.

the convolutional block and has its own output buffer. The Buffer_Conv, Buffer_G01 and Buffer_G02 are the FIFOs to store the data to wait to add on other branches. The Group points are combined with adders to add the data from each branch. The structure of Group1–3 is similar to Group0, so we will not go into details here.

After Group0–3, an average pooling layer is used consisting of adders and a divider. This layer receives an image of size 7×7. We add the 7 data points of each row first and store them in a FIFO. After that, we add the 7 addition outputs together. Then we use a divider to get the average. In the fully connected layer, 1000 output channels operate in parallel. The result of the 1000 outputs will be stored in RAM. A comparator compares the 1000 results, and the number of the largest one is marked as the final result.

4.3. *Hardware utilization analysis*

After introducing the implementation of the network, we analyze the hardware utilization of the RAMs and DSPs of the whole network, which increases while increasing the number of threads from one to eight.

DSPs are used in the convolutional layer in Layer1, the downsample layers in Group0–3, and the fully connected layer. The usage of DSPs depends on the degree of the parallelism of each layer. We can use Eq. (1) to describe the usage of DSPs.

$$N_{\mathrm{DSP}} = mN_{\mathrm{DSP\_}L1} + nN_{\mathrm{DSP\_}G} + nN_{\mathrm{DSP\_}F}. \tag{1}$$

In this equation, N_{DSP} is the number of all the DSP resources used in this network. $N_{\mathrm{DSP\_}L1}$ is the number of the DSPs used when the degree of the output channel parallelism is 1 in Layer1. $N_{\mathrm{DSP\_}G}$ and $N_{\mathrm{DSP\_}F}$ are the numbers of the DSPs used when the degree of the input channel parallelism is 1 in the Group0–3 and the fully connected layer. In the equation, m represents the degree of parallelism of the output channels in Layer1, while n represents the degree of parallelism of the input channels in Group0–3 and the fully connected layer.

As we introduced in Sec. 4.2, the input channels of the convolutional layer in Layer1 and the output channels of the convolutional layers in Group0–3 are fully parallel no matter how many threads we use. From one thread to four threads, only the degree of the output channel parallelism in Layer1 increases, while the degree of the input channel parallelism stays the same in the layers of Group0–3 and the fully connected layer. This means the number m increases from 1 to 4 while n always stays 1 when the threads increase from 1 to 4. However, when the number of threads increases from 4 to 8 or more, not only the degree of the output channel parallelism in Layer1 but also the degree of the parallelism in Groups and the fully connected layer increases. Therefore, m is 8, and n is 2 when using eight threads. Every time the threads double after 8, m and n both double. From this, we can see that from using one thread to four threads, only m increases, so the increase of the DSPs remains low. When using eight threads or more, the total number of DSPs doubles.

The usage of RAMs in this paper is based on the buffers to store parameters, the buffers to store the input image, and the buffers to store intermediate results in Layer1 and in Group0–3. The buffers in Layer1 can be divided into the buffers preparing for the convolutional and max pooling operation and the buffers for the results. The buffers in Group0–3 include not only the former three kinds of buffers but also the buffers preparing for the downsample layer and the buffers to store the results to wait for the other branches. Equations (2) to (4) describe the usage of the buffers.

$$N_{\text{Buff}} = N_{\text{Buff\_}P} + N_{\text{Buff\_}I} + N_{\text{Buff\_}L1} + N_{\text{Buff\_}G}, \tag{2}$$

$$N_{\text{Buff\_}L1} = N_{\text{Buff\_}L1\_C} + N_{\text{Buff\_}L1\_M} + N_{\text{Buff\_}L1\_R}, \tag{3}$$

$$N_{\text{Buff\_}G} = N_{\text{Buff\_}G\_C} + N_{\text{Buff\_}G\_M} + N_{\text{Buff\_}G\_D} + N_{\text{Buff\_}G\_B} + N_{\text{Buff\_}G\_R}. \tag{4}$$

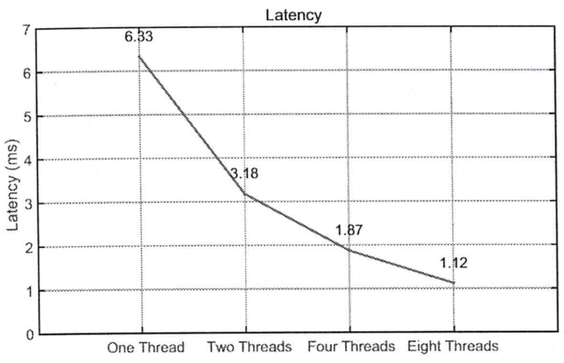
When the threads increase from 1 to 2, only the number of the buffers for the Conv1 Layer, the Maxpool1 Layer and the results in Layer1 increase, while others stay the same. When the threads increase from 2 to 4, not only the buffers in Layer1 increase but also the buffers for the first binary convolutional layers and the downsample layers in Group1–3 increase. In Group1–3, the number of the output channels in the first binary convolutional layer and the downsample layer is twice that of the input channels, as shown in Fig. 2. In this way, the time consumption to transfer the output data to the next stage is twice the time to do the convolution operation, which is the same as the time that it takes for the input features to flow in. Double buffers are needed to store these results to prevent the data in the output buffers from being refreshed by the next row before it can be transferred to the next stage. When the threads increase to 8 or more, one buffer needs to be divided into two buffers in Layer1 and Group0–3, which may cause the RAM utilization to increase after synthesis. The buffers mentioned in this section are usually synthesized into BRAMs. However, when the buffers are very small, the EDA tools may synthesize them into LUTRAMs instead of BRAMs.

5. Experimental Results

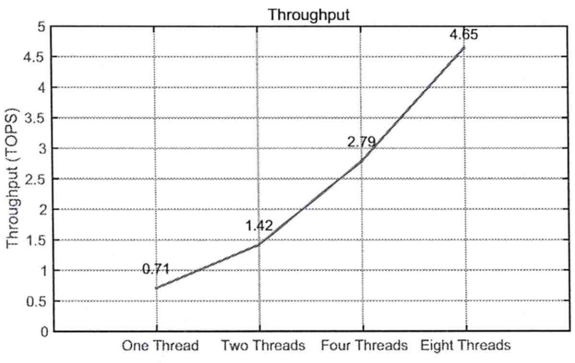
In this section, we show the hardware implementation results of the NASB-ResNet18. We also present the trade-off between hardware utilization and performance. Finally, we compare our results with other state-of-the-art networks.

5.1. *Results of NASB-ResNet18*

The experimental setup used in this paper to perform the measurements consists of an Inspur FP5290G2 system with a dual-socket POWER9 Lagrange 22-core CPU and OpenCAPI interface to an Aphadata ADM-PCIE-9H7 FPGA board with a Xilinx XCVU37P chip. Our network design runs at a 200 MHz clock frequency. We use the ImageNet dataset for the inference measurements in this section.



(a) The latency of different thread methods



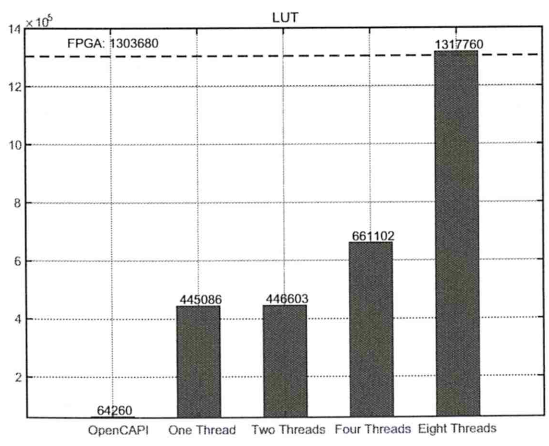
(b) The throughput of different thread methods

Fig. 9. The latency and throughput for different thread counts.

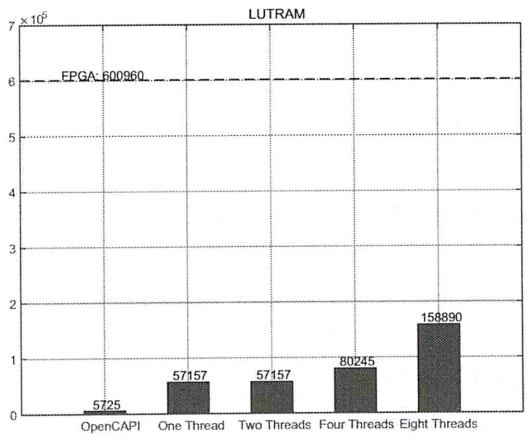
As discussed in Sec. 3, the Top-1 accuracy of this design on the ImageNet dataset can achieve 60.5%, which is almost as high as the accuracy of the full precision ResNet18.

In this paper, we investigate different levels of parallelization to implement the network, using one, two, four and eight threads for Layer1, as illustrated in Sec. 4.2. The latency dependence on parallelization is shown in Fig. 9(a), which is 6.33, 3.18, 1.87 and 1.12 ms for 1, 2, 4 and 8, respectively, with a 200 MHz clock frequency for each ImageNet image. In addition, following the throughput calculation methods in Ref. 27, the throughput of the four methods is 0.71, 1.42, 2.79 and 4.65 TOPS (tera operations per second), as shown in Fig. 9(b). As shown in Fig. 9, the latency decreases by about 50%, and the throughput almost doubles with every doubling of the thread count.

The hardware utilization of the four different methods is shown in Fig. 10. The dashed horizontal line highlighted with the label FPGA represents the available resources on the target FPGA. There is a significant increase in LUTs from 445 K to 1318 K as we scale from one thread to eight threads. The same is true

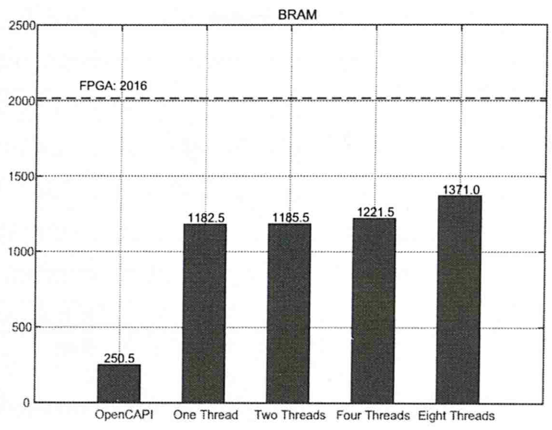


(a) LUT Utilization

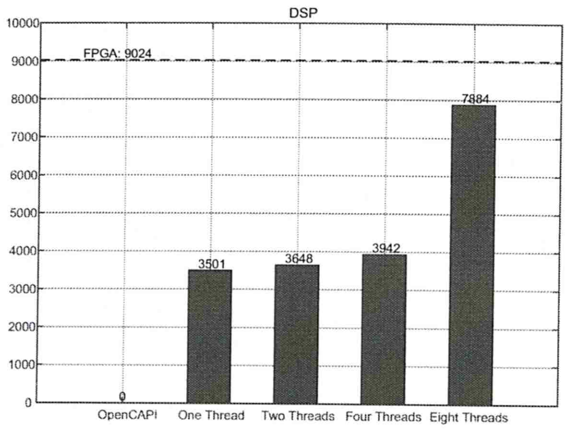


(b) LUTRAM Utilization

Fig. 10. Hardware utilization of FPGA for the full design.



(c) BRAM Utilization



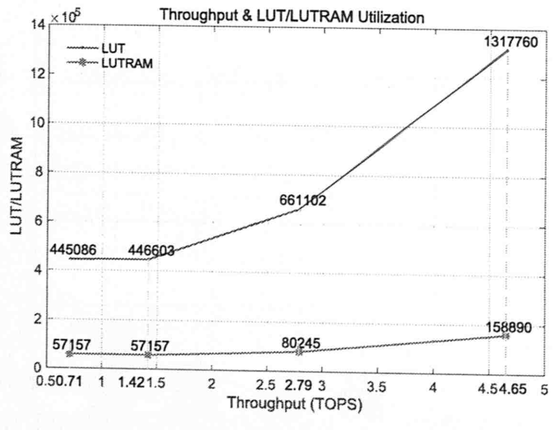
(d) DSP Utilization

Fig. 10. (*Continued*)

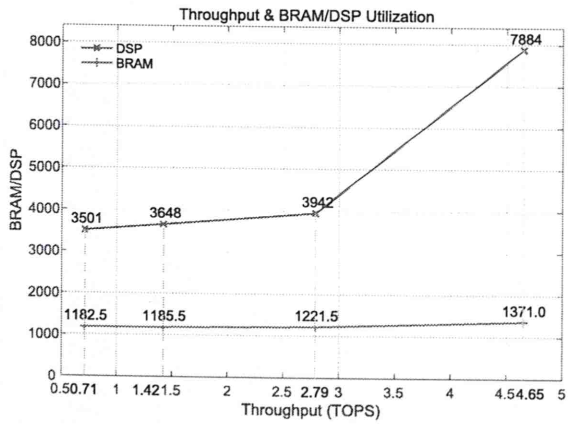
for LUTRAMs as they scale from 57 K to 159 K, though they do not represent a resource bottleneck for the target FPGA. 1600 LUTRAMs are used to store parameters, and the rest are used in the downsample layers. For all thread counts, 44 BRAMs are used to load images, and 1072 BRAMs to store parameters, while 66.5, 69.5, 105.5 and 255 BRAMs are used to store intermediate results for Layer1 and Group0–3, respectively. For DSPs, as illustrated in Sec. 4.3, the results show that the number of $N_{\mathrm{DSP\_}L1}$ is 148 DSP blocks, $N_{\mathrm{DSP\_}G}$ is 1353 and $N_{\mathrm{DSP\_}F}$ is 2000. With this data and the data shown in Fig. 10(d), we can see that the usage of the DSPs meets Eq. (1). Meanwhile, the OpenCAPI interface consumes 64,260 LUTs, 66,233 FFs and 250.5 BRAMs.

5.2. *Hardware utilization versus performance tradeoffs*

Using different number of threads to implement NASB-ResNet18 will impact the latency and throughput performance results as well as the hardware utilization



(a) The throughput and LUT/LUTRAM utilization



(b) The throughput and BRAM/DSP utilization

Fig. 11. The throughput and hardware utilization for different threads.

numbers. Figure 11 shows the throughput versus hardware utilization for the four different thread number cases. The measurements on each line represent the results of one thread, two threads, four threads and eight threads, respectively. Figure 11(a) shows the hardware utilization increase for the LUT and LUTRAM while Fig. 11(b) shows the hardware utilization increase for the DSP and BRAM for different throughput values. The figures show that by increasing the number of threads from one to two, the throughput increases by a factor of 2, at the cost of marginal increases in LUT utilization (0.3%), DSP utilization (4.2%), BRAM utilization (0.3%) while the LUTRAM utilization stays the same. By increasing the number of threads from two to four, the throughput increases by an additional 96.5%, while the LUT utilization increases by 48.0%, the LUTRAM utilization increases by 40.4%, the DSP utilization increases by 8.1% and the BRAM utilization increases by 3.0%. Going from four threads to eight threads, the throughput increases by 66.6%, while the LUT utilization increases by 99.3%, the LUTRAM utilization increases by 98.0%, the DSP utilization increases by 2 times and the BRAM utilization increases by 12.2%.

This analysis shows that using two threads to implement the model has limited impact on hardware resources compared to one thread, but results in doubling the throughput. This indicates that using two threads for input channels in Layer1 is more optimal than one thread for this fully pipelined and semi-parallel channel hardware structure. In addition, we can get further benefit from implementing multiple threads for Layer1. However, the increase in hardware utilization between eight threads and four threads is more pronounced than between four threads and two threads, while the increase in throughput between eight threads and four threads is less pronounced than between four threads and two threads. However, the throughput increase between eight threads and four threads is still large. So the number of threads to use can be decided by the performance requirement and the hardware resources available on the target FPGA.

5.3. *Comparison with other solutions*

We compare our results with other state-of-the-arts hardware implementations of ResNet18. Table 2 shows the comparison of performances between published results and this paper. Compared to Yang *et al.*[28] our implementation with two threads has one-third the latency, while our implementation only uses about two times more hardware resources. Although Yang *et al.*[28] achieve higher model accuracy, this is done using 8-bit data precision (instead of 1 in our case). Further measurements show that our method can achieve an accuracy of up to 67.4% using 8-bit data precision (at the expense of more hardware utilization). Our highest throughput is 7.3× higher than the result of 383.05 GOPS in Kala *et al.*[29] because we pipeline between layers rather than pipeline only inside layers. At the same time, our throughput using similar hardware resources is 3.7× higher than the throughput in Kala *et al.*[29] In comparison with Baskin *et al.*[30] we achieved up to 14.4× lower

Table 2. Comparison with other ResNet18 hardware implementations.

Reference	Yang *et al.*[28]	Kala and Nalesh[29]	Baskin *et al.*[30]	Qu *et al.*[31]
Dataset	ImageNet	—	ImageNet	—
Data precision (bit)	8	2	1	16
Accuracy	66.78%	—	57.5%	—
FPGA platform	ZCU102	Virtex 7XC7VX690T	Stratix 5SGSD8	KCU1500
Frequency (MHz)	166	200	105	240
Latency (ms)	9	—	16.1	14.78
Throughput (TOPS)	—	0.38	—	2.44
LUT	204496	468000	596081	212455
DSP	517	1436	—	5367
BRAM	739	1465	30854 Kbits	1860
Reference	**This work**	**This work**	**This work**	**This work**
Thread	1	2	4	8
Dataset	ImageNet	ImageNet	ImageNet	ImageNet
Data precision (bit)	1	1	1	1
Accuracy	60.5%	60.5%	60.5%	60.5%
FPGA platform	Aphadata ADM-PCIE-9H	Aphadata ADM-PCIE-9H	Aphadata ADM-PCIE-9H	Aphadata ADM-PCIE-9H
Frequency (MHz)	200	200	200	200
Latency (ms)	6.33	3.18	1.87	1.12
Throughput (TOPS)	0.71	1.42	2.79	4.65
LUT	445086	446603	661102	1317760
DSP	3501	3648	3942	7784
BRAM	1182.5	1185.5	1221.5	1371

latency because of the row-channel data structure we proposed and the output channel parallelization instead of the input channel parallelization they use, which leads to a higher degree of parallelism. Our accuracy is also 3% higher than their accuracy with the same bit width, which shows that the NASB strategy results in increasing accuracy. Our implementation with four threads has a 16.3% higher throughput and an eight times lower latency than Qu *et al.*[31] at the expense of using more LUTs but fewer DSPs and BRAMs. Compared to these solutions, our proposed method also demonstrates better flexibility for the trade-off of latency or throughput and hardware utilization.

6. Conclusion

In this paper, we present a low latency and high accuracy hardware implementation for inference on a binarized ResNet18 network on FPGAs. The weights of the convolutional kernels of the main branches are binary, and a neural architecture search strategy is applied to improve the accuracy of the network. A pipeline structure is used between layers to enable parallel processing of multiple layers. A parallel strategy is applied to reduce the transmission time difference of the output results of different layers. In addition, the use of a row-channel data structure prevents the need to fully parallelize the layers for a pipeline between layers, which can achieve a trade-off between the latency and hardware utilization. The implementation leverages the OpenCAPI data interface on a POWER9 system to an Alphadata 9H7 FPGA. For the ImageNet dataset, the inference latency varies from 1.12 to 6.33 ms, and the throughput varies from 4.56 to 0.71 TOPS based on different parallelization strategies, with a Top-1 accuracy of 60.5% and a clock frequency of 200 MHz. We achieve up to 8$\times$ reduction in latency and a 1.9$\times$ improvement in throughput compared to the best previous work. The methods proposed in this paper can also be used to improve the performance of other convolutional neural networks. The code for our implementation is open-source and publicly available on GitHub at https://github.com/MFJI/NASBRESNET.

ORCID

Mengfei Ji https://orcid.org/0000-0001-6151-4046

References

1. S. H. Wang, J. D. Sun, I. Mehmood, C. C. Pan, Y. Chen and Y. D. Zhang, Cerebral micro-bleeding identification based on a nine-layer convolutional neural network with stochastic pooling, *Concurr. Comput. Pract. Exp.* **32** (2020) e5130.
2. L. Alzubaidi, J. L. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaria, M. A. Fadhel, M. Al-Amidie and L. Farhan, Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions, *J. Big Data* **8** (2021) 53.
3. S. H. Gao, M. M. Cheng, K. Zhao, X. Y. Zhang, M. H. Yang and P. Torr, Res2Net: A new multi-scale backbone architecture, *IEEE Trans. Pattern Anal. Mach. Intell.* **43** (2021) 652–662.
4. M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv and Y. Bengio, Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1 (2016), arXiv:1602.02830.
5. Z. C. Liu, W. H. Luo, B. Y. Wu, X. Yang, W. Liu and K. T. Cheng, Bi-Real Net: Binarizing deep network towards real-network performance, *Int. J. Comput. Vis.* **128** (2020) 202–219.
6. B. Zhu, H. P. Hofstee, J. Lee and Z. Al-Ars, Improving gradient paths for binary convolutional neural networks,*33rd British Machine Vision Conf. 2022, BMVC 2022*, London, UK, 21–24 November 2022 (BMVA Press), p. 281.
7. M. Rastegari, V. Ordonez, J. Redmon and A. Farhadi, XNOR-Net: ImageNet classification using binary convolutional neural networks, *14th European Conf. Computer Vision (ECCV)*, (Springer, 2016), pp. 525–542.
8. Y. Bengio, N. Lonard and A. Courville, Estimating or propagating gradients through stochastic neurons for conditional computation (2013), arXiv:1308.3432.
9. B. Z. Zhu, Z. Al-Ars and W. Pan, Towards lossless binary convolutional neural networks using piecewise approximation, *24th European Conf. Artificial Intelligence (ECAI)* (IOS Press, 2020), pp. 1730–1737.
10. T. Elsken, J. H. Metzen and F. Hutter, Neural architecture search: A survey, *J. Mach. Learn. Res.* **20** (2019) 1997–2017.
11. B. Zoph, V. Vasudevan, J. Shlens and Q. V. Le, Learning transferable architectures for scalable image recognition, *31st IEEE/CVF Conf. Computer Vision and Pattern Recognition (CVPR)*, (IEEE, 2018), pp. 8697–8710.

12. B. Z. Zhu, Z. Al-Ars and H. P. Hofstee, NASB: Neural architecture search for binary convolutional neural networks, *Int. Joint Conf. Neural Networks (IJCNN)* held as part of the *IEEE World Congress on Computational Intelligence (IEEE WCCI)*, (IEEE, 2020), pp. 1–8.
13. Y. F. Yang, Q. J. Huang, B. C. Wu, T. J. Zhang, L. Ma, G. Gambardella, M. Blott, L. Lavagno, K. Vissers, J. Wawrzynek and K. Keutzer, ACM, Synetgy: Algorithm-hardware co-design for ConvNet accelerators on embedded FPGAs, *ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, (ACM, 2019), pp. 23–32.
14. K. M. He, X. Y. Zhang, S. Q. Ren and J. Sun, Deep residual learning for image recognition, *2016 IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, Seattle, WA, 2016, pp. 770–778.
15. J. Hu, L. Shen, S. Albanie, G. Sun and E. H. Wu, Squeeze-and-excitation networks, *IEEE Trans. Pattern Anal. Mach. Intell.* **42** (2020) 2011–2023.
16. H. M. Li, X. T. Fan, L. Jiao, W. Cao, X. G. Zhou and L. L. Wang, A high performance FPGA-based accelerator for large-scale convolutional neural networks, *26th Int. Conf. Field-Programmable Logic and Applications (FPL)*, (IEEE, 2016), pp. 1–9.
17. X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang and J. Cong, Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs, *Proc. 54th Annual Design Automation Conf.* (ACM, 2017), pp. 1–6.
18. X. C. Wei, Y. Liang, X. H. Li, O. H. Yu, P. Zhang, A. O. Cong and M. Assoc Comp, TGPA: Tile-Grained pipeline architecture for low latency CNN inference, *37th IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, (IEEE/ACM, 2018), pp. 1–8.
19. S. I. Venieris and C. S. Bouganis, fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs, *24th IEEE Int. Symp. Field-Programmable Custom Computing Machines (FCCM)*, (IEEE, 2016), pp. 40–47.
20. S. Han, X. Y. Liu, H. Z. Mao, J. Pu, A. Pedram, M. A. Horowitz and W. J. Dally, EIE: Efficient inference engine on compressed deep neural network, *43rd ACM/IEEE Annual Int. Symp. Computer Architecture (ISCA)*, (IEEE Press, 2016), pp. 243–254.
21. M. A. Carreira-Perpinan and Y. Idelbayev, “Learning-Compression” algorithms for neural net pruning, *31st IEEE/CVF Conf. Computer Vision and Pattern Recognition (CVPR)*, (IEEE, 2018), pp. 8532–8541.
22. C. Wu, V. Fresse, B. Suffran and H. Konik, Accelerating DNNs from local to virtualized FPGA in the Cloud: A survey of trends, *J. Syst. Archit.* **119** (2021) 102257.
23. R. Andri, L. Cavigelli, D. Rossi and L. Benini, YodaNN(1): An ultra-low power convolutional neural network accelerator based on binary weights, *IEEE-Computer-Society Annual Symp. VLSI (ISVLSI)*, (IEEE, 2016), pp. 236–241.
24. S. Liang, S.Y. Yin, L.B. Liu, W. Luk and S. J. Wei, FP-BNN: Binarized neural network on FPGA, *Neurocomputing* **275** (2018) 1072–1086.
25. Z. Han, J. F. Jiang, J. W. Xu, P. Zhang, X. Q. Zhao, D. Wen and Y. Dou, A high-throughput scalable BNN accelerator with fully pipelined architecture, *CCF Trans. High Perform. Comput.* **3** (2021) 17–30.
26. B. Z. Zhu, P. Hofstee, J. Lee and Z. Al-Ars, Sofar: Shortcut-based fractal architectures for binary convolutional neural networks (2020), arXiv:2009.05317.
27. P. Molchanov, S. Tyree, T. Karras, T. Aila and J. Kautz, Pruning convolutional neural networks for resource efficient inference (2016), arXiv:1611.06440.

28. T. Yang, Y. K. Liao, J. P. Shi, Y. Liang, N. F. Jing and L. Jiang, A Winograd-based CNN accelerator with a fine-grained regular sparsity pattern, *30th Int. Conf. Field-Programmable Logic and Applications (FPL)* (IEEE, 2020), pp. 254–261.
29. S. Kala and S. Nalesh, Efficient CNN accelerator on FPGA, *IETE J. Res.* **66** (2020) 733–740.
30. C. Baskin, E. Zheltonozhskii, A. M. Bronstein, A. Mendelson and N. Liss, Streaming architecture for large-scale quantized neural networks on an FPGA-based dataflow platform, *27th Int. Heterogeneity in Computing Workshop in conjunction with 32nd IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, Vancouver, Canada, 21–25 May 2018, pp. 162–169.
31. X. Y. Qu, Z. H. Huang, Y. Xu, N. Mao, G. Cai and Z. Fang, Cheetah: An accurate assessment mechanism and a high-throughput acceleration architecture oriented toward resource efficiency, *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* **40** (2021) 878–891.