BJTU BNN Accelerator

Introduction

Convolutional neural networks (CNNs) have been employed in many applications, such as image classification, video analysis and speech recognition. However, CNNs are generally compute-intensive and traditional CPU-based hardware platform cannot perform the CNN computation efficiently since the architecture of CPU is not suitable for this computational pattern. Therefore, many other hardware processors are used to accelerate the computation, such as the graphics process unit (GPU), field-programmable gate array (FPGA), etc. Besides using accelerators, reducing the computational complexity and memory requirements is also a method to accelerate the CNN forward propagation. For instance, pruning and quantization are commonly used methods, what's more, binary neural network (BNN) has become a popular research field recently. Unlike the conventional CNN models, BNN adopts XNOR and popcount as the basic operation. XNOR and popcount operations can be implemented using the logical components, which enables some embedded FPGA with few DSP to do neural network computation. Besides, the weights and activations are represented by only one bit, this can relieve the memory access pressure, and further improve the total throughput of the accelerator.

There was an obvious performance gap between the 1-bit and real-valued CNN modules until the IR-Net [1] proposed. IR-Net reduces the accuracy difference using Libra-PB and EDE methods, and full precision shortcut is also used. However, there are also many challenges when implementing IR-Net on the FPGA. Firstly, there are 21 convolution/FC layers in the IR-Net, it can't be implemented on the conventional streaming BNN accelerators. Secondly, the data in the shortcut layer are not binarized, so data should be stored using the global memory whose bandwidth is a challenge in accelerator design. Thirdly, the weights are represented by the product of a scaling factor and a binary kernel, which introduces floating point calculations. Fourthly, BN is used to adjust the distribution of activations, which observably improve the accuracy but also increase the computation.

In this work we design a generic FPGA accelerator design aiming at accelerating BNN neural networks with high performance in accuracy. In this work we only adapt IR-Net on it, we will fit more BNN models in the future. Besides the aforementioned issues are addressed in this work, we also have the following contributions. Firstly, we design an efficient implement of XNOR and popcount operation using the look up table. Secondly, we address the padding issue of the BNN. Thirdly, a deep pipeline structure is formed between kernels, which release the memory access pressure and improve the overall performance. The FPGA design was based on our open-source work of PipeCNN, and was fully modified and optimized to fit the Xilinx Vitis design flow. The hardware platform used was Alveo U50 Acceleration card.

1. Network architecture and optimization

We adapt the IR-Net on the accelerator in this competition because IR-Net has relatively high accuracy in BNNs. In many BNN models, the weights are represented by the product of the binary

weights and a scaling factor to reduce the quantization error and BN is used to adjust the distribution of the activation to improve the accuracy. The experiment shows that the network with the BN layer has better performance. But the BN layer introduces more multiplication operations, which leads to consuming more DSPs. For example, FP-BNN uses DSP for multiplication to calculate BN. In this work, we merge the BN with the convolution operation.

Although the number of multiplications is reduced by merging BN with convolution, the scaling factor and bias are both floating-point numbers. Floating-point calculation consumes a lot of resources on FPGA, so we carry out 8-bit quantification on the activation, scaling factor, bias and other float parameters.

We implement the Xilinx TQT quantization strategy on the IR-Net. When training the truncation value T using the TQT method, the selection of a reasonable initial value is very important. So, we initialize the truncation value T by using TensorRt quantization method. Retraining the model for 10 epochs can get a quantized model with the accuracy loss within 1%.

2. Hardware design

2.1Deep pipeline architecture

The FPGA neural network accelerators can be roughly divided into two categories: a. a single processing engine which processes each layer sequentially; b. a streaming architecture where each network layer corresponds to a compute unit in the accelerator. Streaming accelerators usually have restrictions on the number of network layers, because the hardware demand increase linearly with the number of layers. In order to make it free from the limitation of the network layer number, we chose the former architecture. The top view of the BNN accelerator is shown in the Fig. 1, where a deep pipeline is formed by connecting a few kernels using OpenCL pipes. This architecture makes the computation more efficiently for the pipeline parallelism, and the memory access pressure is reduced for the less frequent access to global memory.

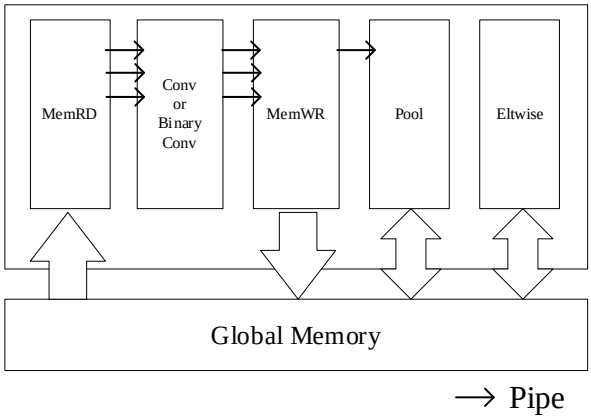

Fig. 1 The top-level of accelerator.

2.2Parallelism selection

There are many calculations can be done parallelly in the convolution operation. In this work, we utilize two parallelisms: the parallelism between each convolution kernels, and the parallelism within one kernel. The parallelism in convolution operations is shown in Fig. 2, a total of $LANE\_NUM$ kernels are doing the convolution calculation parallelly, and in each kernel $8 \times VEC\_SIZE$ bits of weights and activation are XNORed parallelly. So, when the accelerator is well pipelined, a total of $8 \times VEC\_SIZE \times LANE\_NUM$ calculation can be done in each clock.

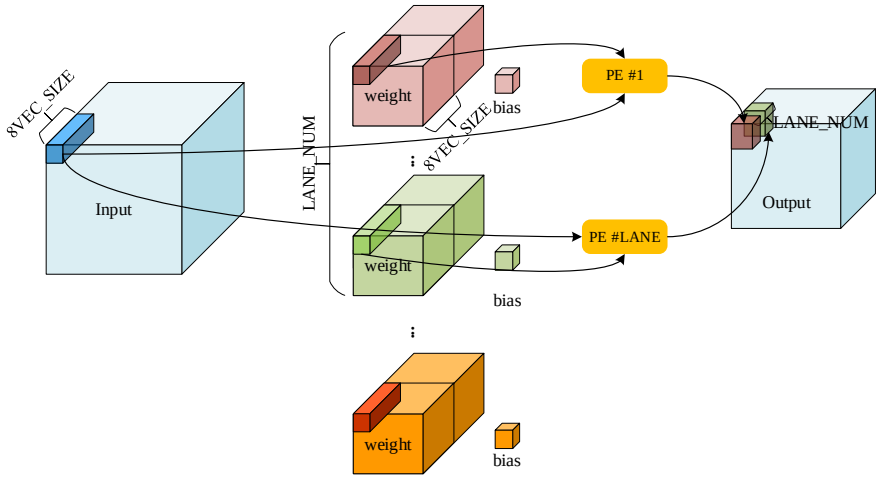

Fig. 2 Parallelism in the binary convolution.

2.3Basic operation in BNN

As shown in Fig. 3, the weights and activations of BNN are represented by either -1 or +1, and they are multiplied and summed. In FPGA based BNN accelerators, the weights and activations are represented by 0 or 1, and the MAC operation is replaced by the XNOR and popcount. The comparison between the element-wise multiplication and XNOR operation is shown in Fig. 4. In Fig. 4 (a), the element-wise multiplication is conducted and then a sum operation shown in equation (1) is conducted, where $NUM_{(+1)}$ and $NUM_{(-1)}$ denotes the number of $+1$ and -1 respectively. In Fig. 4 (b), the element-wise XNOR operation is conducted and then a popcount shown in equation (2) is carried out. Moreover, the equation (2) can be simplified as $2 \times POP(x) - N$, where N denotes the number of total bits.

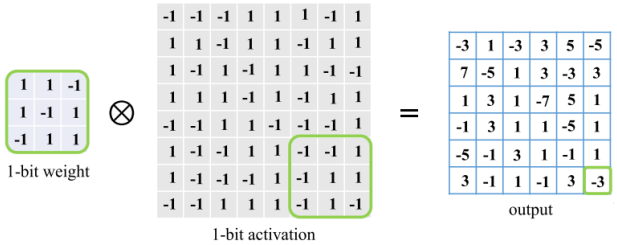

Fig. 3 Convolution Process of Binary Neural Network.

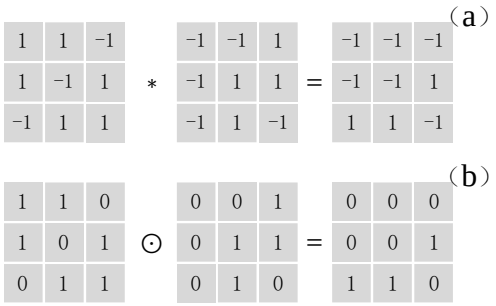

Fig. 4(a) the element-wise multiplication, (b) the XNOR operation.

$$SUM(-1,-1,-1,-1-1,+1,+1,+1,-1) = NUM_{(+1)} - NUM_{(-1)} = -3 \quad (1)$$

$$NUM_{(+1)} - NUM_{(0)} = POP(x) - \big(N - POP(x)\big) = 2 * POP(x) - N = 2 * 3 - 9 = -3 \quad (2)$$

The internal architecture of each PE is shown in Fig. 5, where $8 \times VEC\_SIZE$ bits of activation and weight are sent from the MemRD kernel. Each 8 bits activation and weight are XNORed bitwisely, then a look up table and summation operation are carried out, which realizes the popcount operation efficiently. The convolutional outputs are multiplied with the 8-bits scaling factor and added with the 8-bits bias, so one DSP is used in each PE.

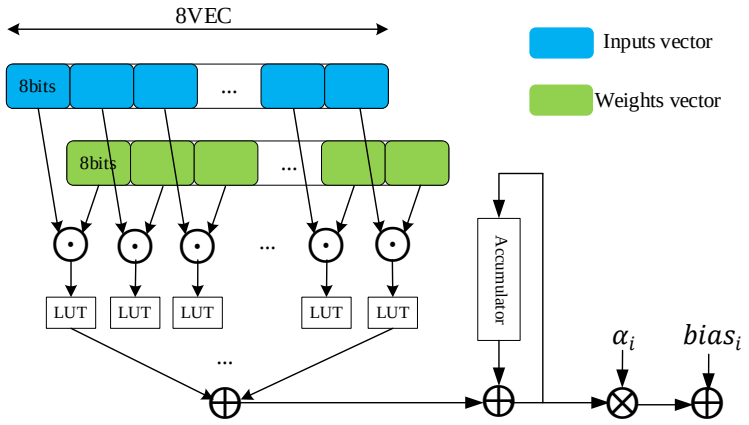

Fig. 5 Architecture of processing element.

2.2Padding in BNN

The activations and weights in BNNs are either $+1$ or -1, but when the padding operation is added, the model is converted to a ternary model, where the activations are $+1,-1$ or 0. There is a prominent difference if the hardware implementation simply fills the padding region with zeros. An example of padding is shown in Fig. 6, where the vector on the left top contains 64 bits, and these bits are all equals to 0.The MemRd kernel judge whether the vector that will be sent to the Binary Conv kernel is a padded vector. If the vector is a padded vector, the MemRd kernel will send a fake weight vector which has 32 bits 0 and 32 bits 1. So, the $XNOR$ and popcount result of the padded vector and fake weight vector equals to 0, and will not introduce error in the result.

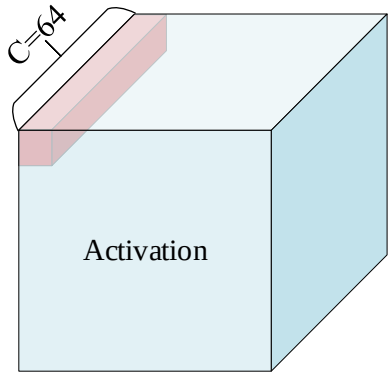

Fig. 6 An example of padding.

3. How to use

1. Set up the Vitis environment, you can follow the UG1393. Our experimental platform is:

```
OS:   ubuntu 18.04 LTS
Tool: Vitis v2020.1 (64-bit)
Card: Alveo U50
BSP:  xilinx_u50_gen3x16_xdma_201920_3
```

2. Clone the repository and download the fixpoint model files from Google Driver or Baidu disk (secret code is 9nc4). Then copy the model file directory to the ***BJTU-BNN-Accelerator/project/data*** directory.

```
$ cd BJTU-BNN-Accelerator/project/data/
$ tree
.
├── IRModel
│   ├── fc1024_bias_summed.dat
│   ├── fc1024bias.dat
│   ├── image.dat
│   ├── scales.dat
│   └── weights.dat
├── README.md
├── imagenet
│   ├── mean_data.dat
│   ├── synset_words.txt
│   └── val.txt
└── picture
    └── cat.jpg
```

3. Go to the project directory, and make.

```
cd BJTU-BNN-Accelerator/project/
make all
```

4. Once the make operation finished, you can run

```
./run.exe conv.xclbin
```

5. Then the log will appear like the following. The result calculated by the accelerator will be verified by comparing with the golden reference. And the time consumption will be reported.

```
************************************************
```

```
PipeCNN: An OpenCL-Based FPGA Accelerator for CNNs
**************************************************

Platform: Xilinx
Totally 1 device(s) are found
  Using Device 0: xilinx_u50_gen3x16_xdma_201920_3
Device OpenCL Version: OpenCL 1.0
Device Max Compute Units: 0
Device Max WorkGroup Size: -1
Device Max WorkItem Size: -1
Device Global Memory Size: 0 MBytes
Device Local Memory Size: 16 KBytes
Device Max Clock Freq: 500 Mhz

11697024 total weights read

3840 total scales read

1024 total fc1024_bias read
1024 total output reference read


150528 bytes image data read from binary files

Executing Layer 1:

Launching single work-item kernel winbuffer

Launching single work-item kernel Conv

Launching single work-item kernel MemWr

Launching single work-item kernel Pool

Executing Layer 2:

Launching single work-item kernel winbuffer

Launching single work-item kernel Binary Conv

Launching single work-item kernel MemWr

Launching single work-item kernel eltwise

Executing Layer 3:

Launching single work-item kernel winbuffer

Launching single work-item kernel Binary Conv

Launching single work-item kernel MemWr

Launching single work-item kernel eltwise

... There are so many lines of log

Executing Layer 21:
```

```
Launching single work-item kernel winbuffer

Launching single work-item kernel Conv

Launching single work-item kernel MemWr

Copyed all batched results from fc_2 buffers.

Copyed all batched results from output_bin buffers.
Selected item = 0 from the combined batch results in fc buffers

Start verifying results ...

Check Pass !!!

The inference result is n01697457 African crocodile, Nile crocodi
le, Crocodylus niloticus   (the prob is 14.00)


PipeCNN exited !!!


------------------

Performance Summary

Kernel runtime summary:
  Layer-1:
    MemRd: 424.154 ms
    Conv : 428.953 ms
    Pool : 418.667 ms
    MemWr: 423.549 ms
    Elt  : 0.000 ms
  Layer-2:
    MemRd: 25.309 ms
    Conv : 25.388 ms
    Pool : 0.000 ms
    MemWr: 22.613 ms
    Elt  : 16.521 ms
  Layer-3:
    MemRd: 23.030 ms
    Conv : 23.124 ms
    Pool : 0.000 ms
    MemWr: 23.114 ms
    Elt  : 16.533 ms

... There are so many lines of log
```

Reference

[1] H. Qin *et al.*, “Forward and Backward Information Retention for Accurate Binary Neural Networks,” in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2020, pp. 2247–2256.