ESE587 Final Project

HuaLing Zheng
112351418
05/15/2020

I. Introduction

A. What is the goal of your project?

The goal is to implement and optimize an parallel vector-dot-product CLP system. Carefully design the architecture and evaluate its performance based on classification with CIFAR-10/100 [1]. First, I want to compare performance of CLP-nano, CLP-Lite and CLP-Parallel. Such as, speed of frequency, usage of BRAM, DSP, LUT, FF and LUTRAM. I will also show the accuracy of floating number calculation in CLP system. Second, I want to compare the performance between regular CLP and pipe-lined CLP. Third, I want to find the bottleneck of data transferring speed of CLP system.

B. Why is this topic important and interesting?

Convolutional neural network (CNN) has been widely employed for image recognition because it can achieve high accuracy by emulating behavior of optic nerves in living creatures [2]. However, deep convolutional neural network relies on GPU to solve a large number of complex operations. In order to implement deep convolutional neural network model on hardware, complex connection relationship and memory usage scheduling are needed [3].

C. Short summary of the results of your project

This report presents the design of FPGA-based accelerator for CNN. The proposed architecture is implemented on MiniZed(a single-core Zynq 7Z007S development board). I used pytorch to train CIFAR 10 and get the inputs, weights, bias and expected outputs data. With one 2d-Conv+ReLU+maxpooling, CLP-Parallel has the best performance in speed and power usage at 62MHz.

II. Background

A. Convolution layer

Figure 1 [4] demonstrates the basic operation of a single channel 2D convolution function. The kernel size is 3×3. The output Y will be calculated by multiplying and adding input mapping X with the same size of kernel W. The X is the input image, W is the weights and Y is the output image. This 2D convolution can also be multi-channel. Figure 2 [4] demonstrates multi-channel 2D convolution. We use a 3D set of weights to produce an output feature map from the entire set of input feature maps [4]. Each output feature maps captures one feature at many locations so that we can use M to capture M different features.

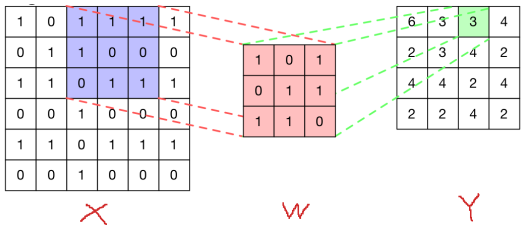

Fig. 1. Single 2D Convolution Operation.

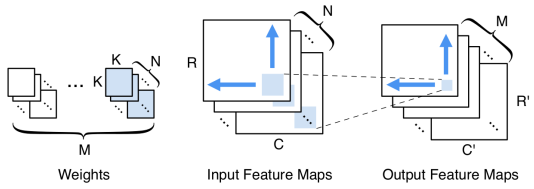

Fig. 2. Multi-Channel 2D Convolution Operation.

B. ReLU Function

The purpose of ReLU function is to make the results of the convolution output nonlinear such that there is no linear relationship between each layer. If there is a linear relationship, the effect of the multi-layer architecture under the deep learning framework will be extremely reduced [5]. In the class, we learned that the common activation functions are sigmoid, tanh and ReLU. The figure 3 [6] showed that how sigmoid and tanh work. However, both sigmoid and tanh require division and exponentiation. It will be a bad idea to build hardware units to do these operations. And we can't let software handle it for us because we would have to shuffle data back and forth between CPU and accelerator too often [6]. Compare to sigmoid and tanh, the ReLU is easy to realize. The piece wise linear nature of ReLU can effectively overcome the gradient vanishing problem [5]. The figure 4 [10] and 5 [6] showed that we can use ReLU approximate sigmoid/tanh functions.

C. Pooling Function

After the convolution and ReLU function, we need pooling function to do down-sampling works. The common pooling functions are average-pooling and max-pooling. Figure 6 [8] demonstrates that how average and max pooling works. Max-pooling function is for maximum values and average pooling is for average values. The pooling function can help to reduce

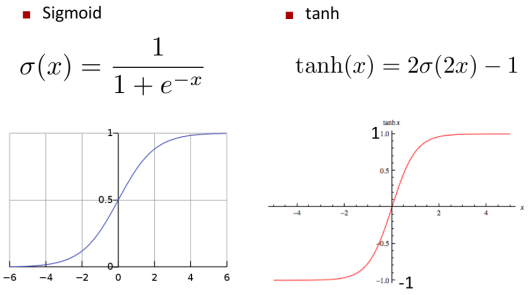

Fig. 3. Sigmoid and Tanh function.

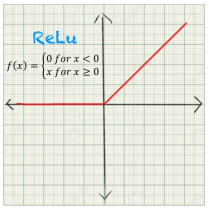

Fig. 4. ReLU Function.

Example: Sigmoid with three segments

$$\sigma_{\text{approx}}(x) = \begin{cases} 0.205x + 0.500, & x < 1.67 \\ 0.075x + 0.717, & 1.67 \leq x < 3.33 \\ 0.017x + 0.910, & x \geq 3.33 \end{cases}$$

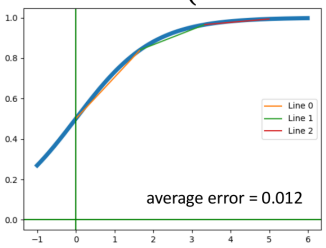

Fig. 5. Piecewise Linear Approximation.

the data size. Thus the number of weights and the amount of calculation will be reduced to avoid the over-fitting [7].

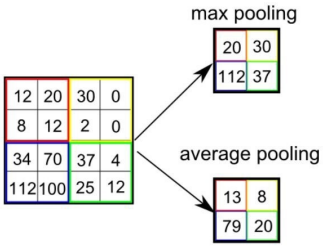

Fig. 6. Common Pooling Function.

D. Architecture of AlexNet

Alexnet is a Deep Convolutional Neural Network (CNN) for image classification that won the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry [9]. The figure 7 [9] showed the detail architecture of AlexNet. In my design, I will use a simplified version of Alexnet to test and evaluate CLP system.

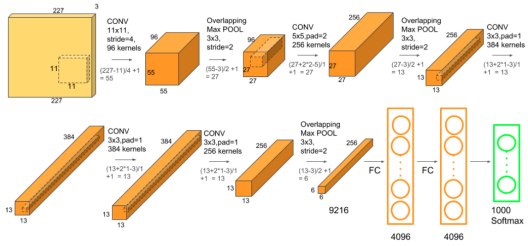

Fig. 7. Architecture of AlexNet.

III. Implementation

In here, I will list the 3 CLP design and compare them in the evaluation part. The first is CLP-nano, this is the most simplified the version of CLP design. Single input come in and single output come out. Simple but not efficeint. Second is the CLP-lite, CLP-lite can tiling the inputs data and save the input Bram size. Third is the CLP-Parallel design. It is the final version of the CLP, it further save the both the input and output Bram size by tiling the N, M and do the parallel computation.

A. CLP-nano Design

Figure 8 [6] shows the basic structure of the CLP-nano. The whole design use three in-chip Bram to take care of the inputs and outputs. The advantage of the CLP-nano is that it is simple to build. However, it is hard to hold entire input feature maps in Bram.

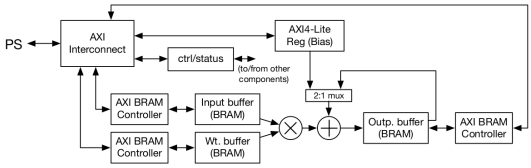

Fig. 8. General Design.

B. CLP-lite Design

CLP-lite is a more flexible design compare to CLP-nano. It will only need to store a fraction of each input and output fearture maps on in-chip Bram. The figure 9 [4] demonstrate the whole process of tiling. The tiling function mainly works on the software part. The hardware structure will not change but instead it will use smaller size Bram to keep the inputs and outputs feature maps.

C. ClP-parallel Design

The parallel design of the CLP system can save even more space for in-chip Bram. As shown in figure 10, the whole input-feature-maps(size $RPrime \times CPrime$), kernels and

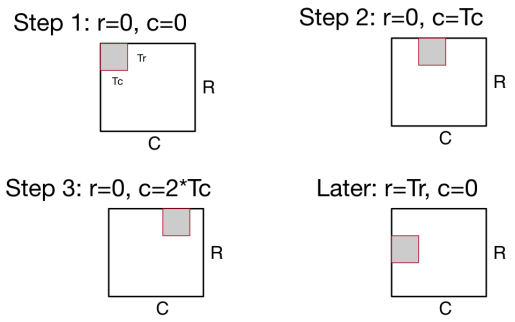

Fig. 9. General Design.

bias are from external memory. Then the input-feature-map will be tiling to small parts(size $TrPrime \times TcPrime$) and send to in-chip memory(Brams). The kernel size(size $K \times K$) will not be tiled and also save to in-chip memory. The bias will be send to in-chip buffers. The tiled output-features(size $Tr \times Tc$) will be saved in the in-chip memory first and send to out-chip memory.

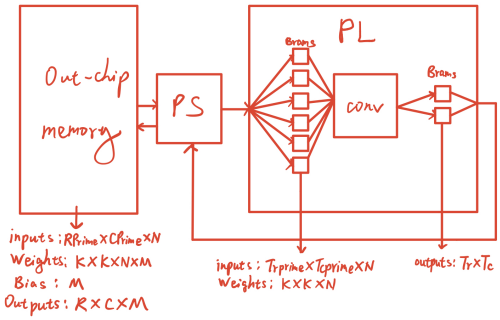

Fig. 10. General Design.

The figure 11 shows the systemVerilog coding to perform parallelism in hardware. In CLP-lite, it can only store one input-feature map at once. In CLP-parallel, we can store $Tm \times Tn \times K \times K$ weights data. In this case, we can realize parallel computation. As shown in Figure 12, this is a two parallel CLP system. Tn=2 and Tm=2. The total Bram usage is 8. The most important thing is that the weights size must fit to Bram. Otherwise, we need to find another way to perform CLP-parallel system. The system will receive 2 sets of inputs data(size $2 \times TrPrime \times TcPrime$). And 2 sets of $2 \times K \times K$ weights data. After that, the 2 parallel system will start to do the multiply and accumulation simultaneously. It generate two output-feature Y0 and Y1. These two output-feature will mapping to the correct position of the outside memory in the software(SDK).

Of course, parallelism is just the beginning of the optimization. The system need piple-line to become more efficient and speedy. I will show the pipelin performance in the next session. Using DMA to transfer inputs data will also improve the CLP-parallel system and I will add that in the future work.

```
if(i<=K-1) begin
    if(j<=(K-1)*4) begin
        if(r<=(TR-1)) begin
            if(c<=(TC-1)*4) begin
                if ((STRIDE_S*r+i < TrPrime) && (((STRIDE_S*c+j)/4) < TcPrime)) begin
                    bram_x0_addr <= c+r*(TrPrime*4)+j+i*(TrPrime*4);
                    bram_x1_addr <= c+r*(TrPrime*4)+j+i*(TrPrime*4);
                    bram_w0_0_addr <= j+i*(K*4);
                    bram_w0_1_addr <= j+i*(K*4);
                    bram_w1_0_addr <= j+i*(K*4);
                    bram_w1_1_addr <= j+i*(K*4);
                    bram_y0_addr <= c+r*(TR*4);
                    bram_y1_addr <= c+r*(TR*4);
                    c <= c+4;
                end
            end
            else begin
                r <= r+1;
                c <= 0;
            end
        end
        else begin
            j <= j+4;
            r <= 0;
            c <= 0;
        end
    end
    else begin
        i <= i+1;
        j <= 0;
        r <= 0;
        c <= 0;
    end
end
else begin
    finish <= 1;
end
```

Fig. 11. General Design.

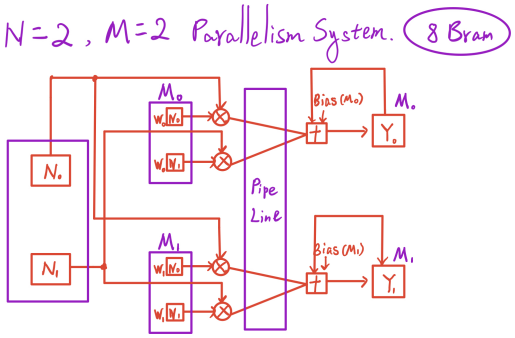

Fig. 12. CLP-Parallel System.

D. Simple Architecture of CNN

To test the whole CLP system, I designed a simple CNN architecture based on Alexnet architecture. The figure 13 and 14 is my design. The first convolutional layer will get 3 channel 32×32 picture inputs. After convolution, ReLU and MaxPooling function, the first layer will have 6 channel 14×14 outputs. The first layer outputs will be recognized as the inputs of the second layer convolution. Finally, I will get 10 channel 5×5 outputs. I use Pytorch to trian a CIFAR10 data based on my simple CNN. After training, I can pull out the inputs, weights, bias and expected outputs data for both convolutional layers.

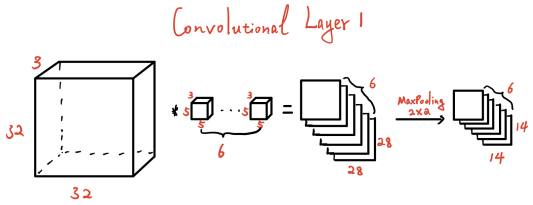

Fig. 13. Simple CNN Layer 1.

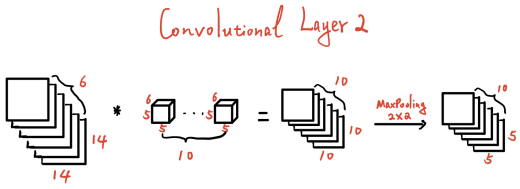

Fig. 14. Simple CNN Layer 2.

IV. Evaluation

I use MiniZed FPGA board as only hardware to perform all the experiments. A Thinkpad PC(with Intel i7-3740QM CPU, 8 cores with 2.7GHz, 16G RAM and 64 bit Ubuntu 18.04 OS) running the Xilinx Vivado and connect MiniZed.

A. Experimental Setup

All the RTL system verilog code for CNN design already been tested in Vivado IDE. I wrote the test-bench to do the behavior stimulation and synthesis test. Based on the waveform, I can check all the control logic work correct or not. And I can also check the timing report and find the critical path with synthesis's report. After that I do implementation which can be used to verify the resource utilization. The usage of LUT, LUTRAM, FF, BRAM, DSP and BUFG can be easily checked.

B. AXI Timer Setup

Based on the timer instruction(example coding file), I added a AXI timer to the design. Figure 15 showed the whole vivado design. However, the AXI timer didn't really work in SDK(I didn't find the reason). So I used "xtime_l.h" to measure the execution time.

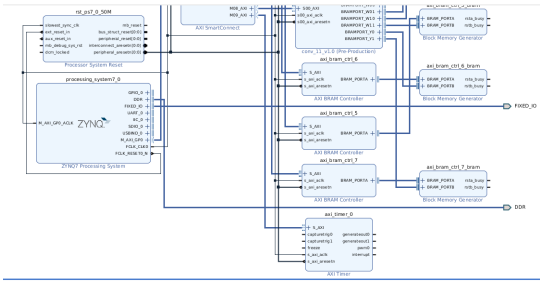

Fig. 15. AXI Timer in CLP-Parallel Design.

C. Testing Results

In the first experiment, I compared the performance of the CLP-parallel, CLP-lite and CLP-nano system with CIFAR10 data on my simple CNN. Figure 16 showed the test results of the performance. Obviously, the CLP-parallel has the best efficiency to use all the resources(with Tr,Tc = 2). And it has the fastest execution time. The second experiment is to find the best performance pipeline numbers and the critical path in the CLP-parallel system. Figure 17 showed that 6 pipeline gives the best efficient performance in the CLP-parallel system. The critical path is from pl_status to add-multiply function. Based on the timer measurement, the bottle-neck is actually at data sending part(PS to Costumed IP). The average data sending cost 47.63us. The figure 18 shows the results.

Metrics	CLP-nano	CLP-lite	CLP-parallel
Frequency	71MHz	71MHz	62MHz
LUT	6638 /46.10%	6647 /46.16%	13730 /96.35%
LUTRAM	961 /16.02%	961 /16.02%	1537 /25.62%
FF	6673 /23.17%	6657 /23.11%	13300 /46.18%
BRAM	48 /96%	12 /24%	32 /64%
DSP	12 /18.18%	12 /18.18%	24 /36.36%
BUFG	1 /3.13%	1 /3.13%	1 /3.13%
Execution Time	921.19 us	15.69 us	12.03 us
Tr,Tc	4	2	2

Fig. 16. Performance Comparison.

CLP- Parallel	No pipeline	1 pipeline	4 pipeline	6 pipeline	8 pipeline	max pipeline (16)
Frequency	16MHz	24MHz	60MHz	62MHz	60MHz	65MHz
LUT	14164 /98.36%	14060 /97.64%	13700 /95.14%	13730 /96.35%	13730 /96.35%	13879 /96.38%
LUTRAM	1537 /25.62%	1537 /25.62%	1537 /25.62%	1537 /25.62%	1574 /26.23%	1712 /28.53%
FF	10964 /38.07%	11254 /39.08%	12454 /43.24%	13300 /46.18%	13816 /47.97%	15896 /55.19%
BRAM	20 /40%	20 /40%	20 /40%	32 /64%	32 /64%	32 /64%
DSP	24 /36.36%	24 /36.36%	24 /36.36%	24 /36.36%	24 /36.36%	24 /36.36%
BUFG	1 /3.13%	1 /3.13%	1 /3.13%	1 /3.13%	1 /3.13%	1 /3.13%

Fig. 17. Pipe-line performance.

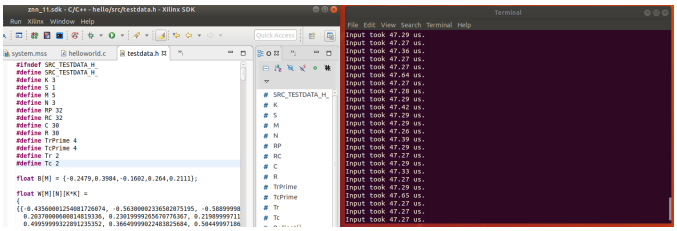

Fig. 18. Bottle-neck.

V. Conclusions and Future Work

In this report, I implement the single layer CNN+ReLU+Maxpooling function on MiniZed developing FPGA board with 32 floating point precision using Xilinx HLS tool. My CLP-parallel system is about 5 times faster than regular CLP-Lite system. My CLP-parallel based on FPGA runs at 62MHz and the power consumption performance better than CLP-Lite and CLP-nano. The whole system is simulated and synthesized in vivado.

The future work will be improve the parallel system. Right now, I am using Brams outside of the custom IP. The LUT usage was huge because the large data transferring between these Brams and PS. To reduce LUP usage and make

parallelism system even larger, I need include these Brams in to a custom IP. And all the data will be tranfer to a big Bram. Inside the costom IP this big Bram will be split to several small brams depend on the address. See the figure 19. The data transfer can use DMA stream to speed up. After these improvement, I believe the CLP-parallel system will get even better performance.

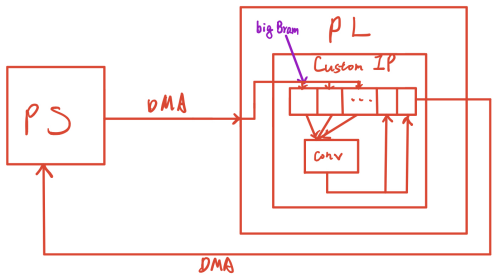

Fig. 19. Future CLP-Parallel System.

REFERENCES

[1] A. Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, Department of Computer Science, University of Toronto, 2009.

[2] Zhang et al., "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," FPGA 2015.

[3] Huynh Vinh Phu, Tran Minh Tan, Phan Van Men, Nguyen Van Hieu, Truong Van Cuong, "Design and Implementation of Configurable Convolutional Neural Network on FPGA", Information and Computer Science (NICS) 2019 6th NAFOSTED Conference on, pp. 298-302, 2019.

[4] Milder, Peter. ESE 587 Class Slide Topic 9.

[5] Maas, Andrew L.; Hannun, Awni Y.; Ng, Andrew Y.(June 2013). "Rectifier nonlinearities improve neural network acoustic models" January 2017.

[6] Milder, Peter. ESE 587 Class Slide Topic 15.

[7] Yongmei Zhou and Jingfei Jiang, "An FPGA-based accelerator implementation for deep convolutional neural networks," 2015 4th International Conference on Computer Science and Network Technology (ICCSNT), Harbin, 2015, pp. 829-832.

[8] Quora:https://www.quora.com/What-is-max-pooling-in-convolutional-neural-networks.

[9] OpenGenus IQ:https://iq.opengenus.org/architecture-and-use-of-alexnet/

[10] Towards Data Science: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6