An FPGA-based Accelerator Implementation for Deep Convolutional Neural Networks

Yongmei Zhou[1]
College of Computer
National University of Defense Technology
Changsha, China
yongmei0102@163.com

Jingfei Jiang[2]
College of Computer
National University of Defense Technology
Changsha, China
jingfeijiang@126.com

***Abstract*—Deep convolutional neural networks (CNN) is highly efficient in image recognition tasks such as MNIST digit recognition. Accelerators based on FPGA platform are proposed since general purpose processor is disappointing in terms of performance when dealing with recognition tasks. Recently, an optimized FPGA-based accelerator design (work 1) has been proposed claiming best performance compared with existing implementations. But as the author acknowledged, performance could be better if fixed point presentation and computation elements had been used. Inspired by its methodology in implementing the Alexnet convolutional neural network, we implement a 5-layer accelerator for MNIST digit recognition task using the same Vivado HLS tool but using 11-bits fixed point precision on a Virtex7 FPGA. We compare performance on FPGA platform with the performance of the target CNN on MATLAB/CPU platform; we reach a speedup of 16.42. Our implementation runs at 150MHz and reaches a peak performance of 16.58 GMACS. Since our target CNN is simpler, we use much less resource than work 1 has used.**

Keywords—FPGA; Convolutional Neural Network; fixed-point arithmetic; HLS

I. INTRODUCTION

Deep convolutional neural networks (CNN) is wildly used in image recognition tasks such as MNIST digit recognition[1]. General purpose processor is not really efficient in dealing with such recognition tasks. An optimized FPGA-based accelerator design[2] targeting at ImageNet classification[3] has been proposed recently which outperforms all previous works[4][5][6][7][8]. Despite its stunning performance, design[2] does not explore the parameter space of fixed point precision, though using fixed point precision is considerably promising as design[2] has pointed out. We also noticed that the Xilinx HLS tool[9][10] design[2] used is highly productive in implementing a deep convolutional neural network. Based on above observations, we design and implement a 5-layer accelerator for MNIST digit database using HLS tool in Vivado 2014.4 system suite and exploring economical fixed point presentation[11][12] without hurting the recognition accuracy on a Virtex7 FPGA. We compare performance on our FPGA platform with the performance of the target CNN in DeepLearnToolbox-master on MATLAB/CPU platform. Our FPGA implementation runs at 150MHz and reaches a peak performance of 16.58 GMACS, which can be easily improved if we design to use more resources. In terms of the running time of processing one input feature map, our work is 16.42 times faster than the MATLAB/CPU code.

II. BACKGROUND

A. CNN Basics

Given a preprocessed natural image (also called an input feature map), a convolutional value is the dot product of a pixel-matrice (part of the input feature map) and a weight-matrice. Figure 1 shows an example of a convolutional layer with one input-feature-map, three weight-matrix(of a size of 2*2) and three output-feature-maps; Code 1 shows how the convolutional operation can be expressed in C code in HLS tool.

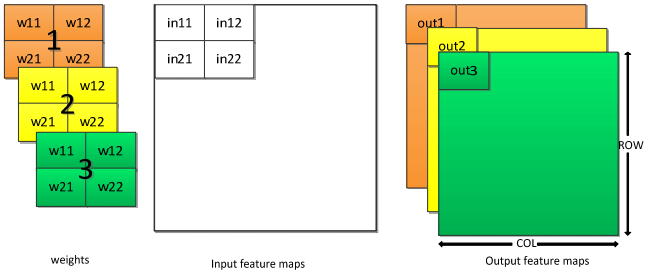

Figure 1: Example of a convolutional layer

```
For (r = 0; r < ROW; r++){
  For (c = 0; c < COL; c++){
    For (out = 0; out < 3; out++){
      output_fm[r][c][out] = w11*in11 + w12*in12
                           + w21*in21 + w22*in22;
}}}
```

Code 1: Example code of a convolutional layer in HLS

B. CNN in MATLAB/CPU platform

Our CNN prototype is as defined in DeepLearnToolbox-master. Figure 2 shows the architecture of our target CNN; it has one input layer, two convolutional layers and two pooling layers.

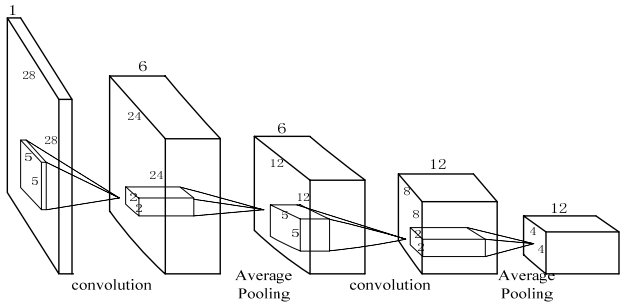

Figure 2: DeepLearnToolbox-master CNN

III. ACCELERATOR DESIGN

A. Design Overview

We decide to use Vivado HLS tool to implement our 5-layer CNN in FPGA platform. As shown in Figure 3, our accelerator requests kernels, biases and input-feature-maps from external memory(such as DDR3) and buffers all these inputs using on_chip memory. Note that the computations of the four layers in Figure 3 are independent of one another.

As shown in Figure 4, a convolutional computation in our design is to perform the dot product of two 5*5 matrix. Code 2 depicts the convolutional computation of layer 2 in HLS. The array w[6][25] accepts initial values through arguments (which will be translated as I/O ports during C synthesis) of the top function(which will be translated as the whole accelerator circuit during C synthesis) and will be implemented as ROM after C synthesis. The array in[5][28] buffers some five lines of input-feature-maps and will be implemented as 5 true-dual-port RAM in a depth of 28 during C synthesis. We use a synthesis directive "#pragma HLS UNROLL" hoping that the C synthesis tool will generate 25 multipliers and thus the generated circuit can perform 25 multiply operations in parallel. But in fact, the C synthesis tool cannot schedule the 25 multiply operation in parallel though it does generate 25 multipliers because of the shared variable result[row][col][ofm]. To eliminate the bad impact from such a shared variable, we rewrite the loop in code 2 and unroll the innermost loop in code 2 by hand. Then the generated circuit of layer 2 will be similar to the one in figure 5.

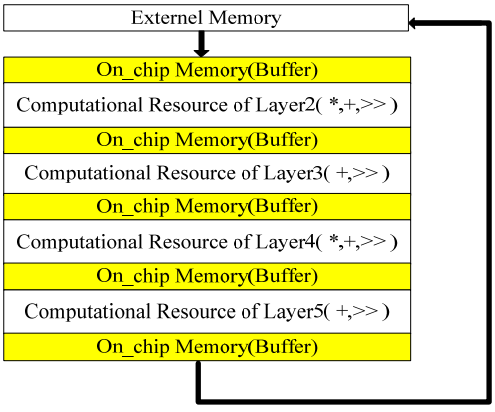

Figure 3: Overview of accelerator design

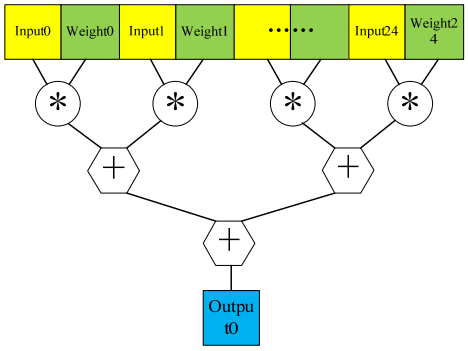

Figure 4: Convolutional computation

```
For (row = 0; row < 24; row++){
  For (col = 0; col < 24; col++){
    For (ofm = 0; ofm < 6; ofm++){
      For (i = 0; i < 25; i++){
#pragma HLS UNROLL
          result[row][col][ofm] = result[row][col][ofm] +
w[ofm][24-i]*in[i/5][col+i%5] ;
 } } } }
```

Code 2: a convolutional computation in HLS

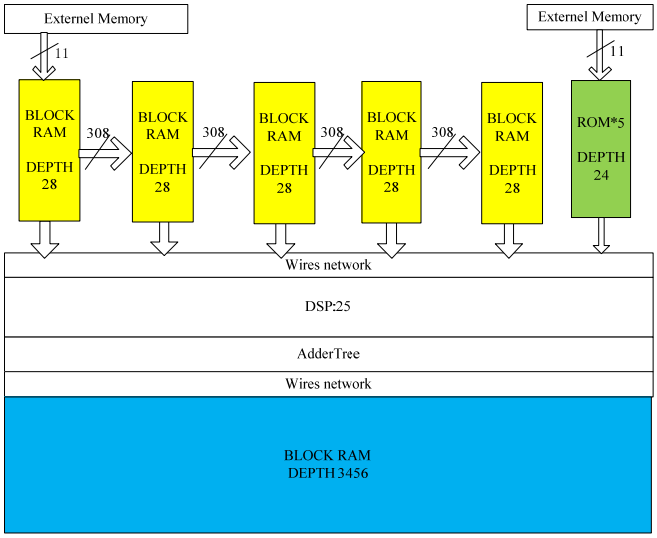

Figure 5： Implementation 1 of layer 2

As shown in Figure 5, we assume that operands are encoded in 11-bits fixed point precision(data_11 in code 5) with 1 sign bit, 4 integer bits and 6 fraction bits; the input-data-port reads in one pixel at a time and this input pixel will be stored in Block RAM; the procedure of newly input value passing through the five Block RAM is depicted by arrows in Figure 5.

Now there is sufficient on-chip memory to store operands in a pixel-matrice and a weight-matrice both in a size of 5*5, and there are enough computation resources to perform 25 multiply and add operations in parallel, but the circuit in Figure 5 can still not perform 25 multiply operations in parallel because all needed operands in one convolutional

computation cannot be read out in one cycle because of the RAM/ROM having only two input/output ports; the C synthesis tool will give a warning in console saying that the tool cannot schedule all load operations in line x because of limited memory ports.

So we get on the track of providing more memory ports in the accelerator. We uses seas of registers to replace the five BRAM thus allowing parallel multiply at last by inserting another synthesis directive called "array partition" into the C loops .

To obtain a better throughput, we think the bottleneck is that we design to stream in only one input value of the input-feature-map in a data request; we can at most generate one output value of the output-feature-map at a time. It means that it takes at least 24*24*6 (3456) cycles for layer 2 to produce

So we get on the track of streaming in more input values instead of just one in the accelerator. We try to stream in one line (28) values of the input-feature-map in a data request and use a wider (28*11bit) data port as shown in Figure 7. With more input values streamed in and stored in registers, we are able to load more operands in parallel for convolutional computation; so we tend to add enough computation resources to perform convolutional computation (many multiply and add operations) in parallel; as a result of that, the circuit can produce in theory one line(24) values of the output-feature-maps. The idea of streaming in many values in one data request(in one cycle if pipelined appropriately) can be realized by concatenating 28 11-bit data to 308-bit before the circuit reading through the input port and later separating 308-bit data to 11-bit each before operating on the data. To concatenating data, we primarily use shift operator and type transformation operator; the similar method is used when separating data.

```
data_11 ifm[5][28];
#pragma HLS ARRAY_PARTITION variable=ifm complete dim=1
#pragma HLS ARRAY_PARTITION variable=ifm complete dim=2

data_11 w[6][25];
#pragma HLS ARRAY_PARTITION variable=w complete dim=1
#pragma HLS ARRAY_PARTITION variable=w complete dim=2

data_11 result[24][24][6];
#pragma HLS ARRAY_PARTITION variable=result complete dim=2
#pragma HLS ARRAY_PARTITION variable=result complete dim=3

For (row = 0; row < 24; row++){
#pragma HLS PIPELINE
  For (col = 0; col < 24; col++){
#pragma HLS UNROLL
    For (ofm = 0; ofm < 6; ofm++){
#pragma HLS UNROLL
 result[row][col][ofm] = ifm[0][col]*w[ofm][24]+ ifm[0][col]*w[ofm][23]+ ⋯\
                       + ifm[1][col]*w[ofm][19]+ ifm[1][col]*w[ofm][18]+ ⋯\
                       + ifm[2][col]*w[ofm][14]+ ifm[2][col]*w[ofm][13]+ ⋯\
                       + ifm[3][col]*w[ofm][ 9 ]+ ifm[3][col]*w[ofm][ 8 ]+ ⋯\
                       + ifm[4][col]*w[ofm][ 4 ]+ ifm[4][col]*w[ofm][ 3 ]+ ⋯ ;
} } }
```

Code 5: array partition, loop unroll and loop pipeline optimization

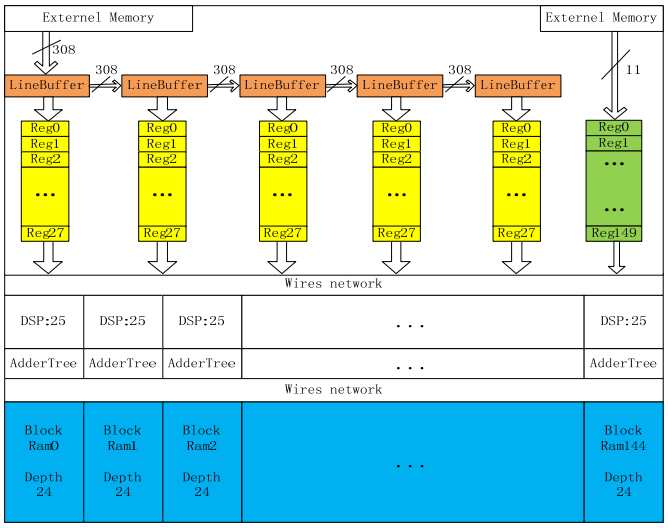

Figure 7 : Implementation 3 of layer 2

IV. Evaluation

We compare FPGA design with the program in matlab on a PC (with Intel i7-4790K CPU, 7.85G RAM and 64 bit windows 7 OS).

A. Experimental Setup

In HLS integrated development environment(IDE), our accelerator program is debugged and synthesized so as to generate RTL Verilog code. The RTL Verilog code is cosimulationed and exported as a packaged IP which can be utilized later in the traditional vivado IDE; a modelsim waveform file and a vivado project are generated as well in this stage. We open the accelerator project in vivado IDE and run the behavior simulation, synthesis and implementation to verify the logic and to report the timing and resource utilization; some results will be presented later in section4.2.

B. Experimental Results

Our first result is about the bit-width of fixed point data precision. We assume that the recognition accuracy is highly related to data precision. To approximate the recognition accuracy of the CNN program in matlab which uses double precision and to at the same time minimize the bit-width of fixed point data in FPGA accelerator design, we explore a limited space of fixed-point precision. The inflection point which has the minimal bit-width without hurting recognition accuracy is a fixed point precision of 11 bits in which 5 bits are used for integer. We test 250 frames of handwriting digit images in the testing stage and the results are shown in Table 2-1 and 2-2.

Table II-1. Exploration of Fixed Point Precision

	Time(ms/frames)	Accuracy (bad/250frames)	Data type
.m file	0.4	8/250	Double
.c file	0.024	8/250	11 bit fixed, 5 bit integer

TABLE II-2. EXPLORATION OF FIXED POINT PRECISION

Data type	Bad (50frames)
<32, 8>	[3, 19, 46]
<20, 6>	[3, 19, 46]
<19, 5>	[3, 19, 46]
<18, 4>	[21, 34, 35, 36, 42, 44, 46]
<18, 5>	[3, 19, 46]
<15, 5>	[3, 19, 46]
<12, 5>	[3, 19, 46]
< 9, 5>	[3, 19, 35, 42, 43]
<11, 5>	[3, 19, 46]
<10, 5>	[3, 19, , 42, 46]

Note: the column titled "bad" shows indexes of differences between computed maximums and actual maximums

The second result is about the clock frequency and resource utilization of our design. When working in HLS IDE, we get estimate reports on frequency and resource utilization as shown in Table 3 and Table 4 when we have done C synthesis(synthesize C design to RTL) and not done RTL synthesis(synthesize RTL design to gates); when working in vivado IDE, we get different reports on frequency(a timing constraint of 6.66ns is met) and resource utilization after synthesizing in vivado as shown in Table 5.

TABLE III. ESTIMATED TIMING IN HLS

Clock(ns)	6.66
Clock cycles	3815
CC of layer2	1288
CC of layer3	13
CC of layer4	1421
CC of layer5	1004
Time(ms)	0.0254

TABLE IV. ESTIMATED RESOURCE UTILIZATION IN HLS

Resource	DSP48E	BRAM	LUT	FF
Used	638	0	66364	51125
Available	2800	2060	303600	607200
Utilization(%)	22	0	26.41	7.6

TABLE V. RESOURCE UTILIZATION IN VIVADO SYNTHESIS REPORT

Resource	DSP	BRAM	Memory LUT	LUT	FF	I/O
Used	83	0	5196	80175	46140	329
Available	2800	2060	130800	303600	607200	700
Utilization(%)	2.96	0	3.97	26.41	7.6	47

The clock frequency is 150 MHz in both HLS and vivado reports. The DSP utilizations of Table 4 and Table 5 are different; this is because during HLS C synthesis stage and vivado RTL synthesis stage, tools do not map a DSP to each multiply operation but use LUT instead as many as possible; when there is an operation like in[i]*w[j] in C code, it is sometimes translated into shift operation and add/subtraction operation like in[i]>>2 - in[i]>>3 in Verilog code.

TABLE VI. ESTIMATED TIMING OF LAYER2 IN HLS

directive	innermost	middle	outermost	Total/cycle
none	29*6	179*24	4311*28	120708
unrolled	--	19*6*24	2828*28	79184
Pipelined in middle loop	--	21+24-2	46*28	1288
Pipelined in outermost loop	--	--	19+28-2	45

V. CONCLUSIONS

In this work, we implement the feedforward CNN function on a xc7vx485tffg1761-2 FPGA in 11-bit fixed point precision using Xilinx HLS tool. Our implementation is about 16.42 times faster than the CNN function on the matlab/PC platform; our accelerator based on FPGA runs at 150MHz(the frequency of PC is 4.00 GHz) and the power consumption is far less than the PC's; our accelerator is simulated in modelsim and synthesized in 2014.4 vivado IDE. But we can't say that there is no room for improvement considering the small resource utilization. In future, we may take into consideration how our accelerator can really work with external memory and take more control on the usage of DSP48E and other resource; we may implement a larger and scalable CNN accelerator which will be more attractive in terms of application.

ACKNOWLEDGMENT

Thanks for my advisor Jingfei Jiang and fellow students. Without their advice, I wouldn't have finished this work.

REFERENCES

[1] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner. Gradient-based learning applied to document recognition. Proceeding of the IEEE, VOL.86, NO.11,NOVEMBER(1998)

[2] C. Zhang, P. li, G. Sun, Y. Guan, B. Xiao and J. Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. FPGA'15, February 22-24, (2015), Monterey, California, USA

[3] A. Krizhevsky, I. Sutskever and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Network. Advances in Neural Information Processing Systems 25. Curran Associates, Inc., (2012)

[4] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf. A programmable parallel accelerator for learning and classification. PACT'10, September 11-15, (2010), Vienna, Austria.

[5] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi. A dynamically configurable coprocessor for convolutional neural networks. ISCA'10, June 19‐23, (2010), Saint-Malo, France.

[6] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. Cnp: An fpga-based processor for convolutional networks. Field Programmable Logic and Applications, (2009).

[7] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf. A massively parallel coprocessor for convolutional neural networks. Application-specific Systems, Architectures and Processors, (2009).

[8] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal. Memory-centric accelerator design for convolutional neural networks. International Conference on Computer Design, (2013), Saint-Malo, France.

[9] An Independent Evaluation of: The AutoESL AutoPilot High-Level Synthesis Tool. Berkeley Design Technology, Inc. (2010).

[10] Vivado Design Suite User Guide High-Level Synthesis. October 1, (2014).

[11] J. Jiang, R. Hu, M. Lujάn, Y. Dou. Accuracy Evaluation of Deep Belief Networks with Fixed-Point Arithmetic. Computer Modelling and New Technologies. (2014).

[12] S. Gupta, A. Agrawal, K. Gopalakrishnan, P. Narayanan. Deep Learning with Limited Numerical Precision. Proceedings of the 32nd International Conference on Machine Learning, Lille, France, (2015).