ECE1718HS Assignment #2 Report

Group 3

Guoxian Wu
Lingfeng Wu
Haotian Zhu
Zhuojun Yu

Table of Contents:

Software Model

Our objective is to build a hardware inference accelerator. In order to achieve this, we need a compatible software model to simulate and verify the hardware implementation.

1.1 Fixed-point Arithmetic

The original software implementation is taking advantage of the high precision of floating point arithmetics. However, it doesn't seem to be the optimal solution in hardware because of the high data footprint and complex computation, which are expensive regarding hardware area. Moreover, the CNN model could achieve a similar performance of accuracy without high precision of floating point. Thus, we can modify the program into a model using fixed-point arithmetic, test its performance and find out the minimum number of bits needed with acceptable accuracy.

Number of fractional bits vs Accuracy

Number of Fractional Bits	Accuracy
5	0.598
6	0.847
7	0.88
>8	0.9

As in the above table, the accuracy drops significantly when there are only 5 fractional bits, and the model keeps the original accuracy when there are more than 8 fractional bits. Therefore, we decided to keep 8 fractional bits to stay at a higher accuracy without consuming too much resources.

The number of bits for the integer parts is trickier. Most of the variables have a very limited range, and some won't even go beyond 2. But the intermediate results of the convolution layer can be very large and are critical to the final results, so even if they are costly, they have to be kept. For the convolution layer, 39 bits are used to represent the intermediate variables. The weights of the convolution layer, however,

seem to be less critical because the final result of the convolution layer will be fed into a sigmoid function.

Since most of the variables are signed, the final format of our fixed-point arithmetic is signed Q8.8, a total of 16 bits that is also friendly to most software models except Conv layer.

Number of Bits Needed for Integer Parts of Layers

Layer	Number of Bits for Integer Part
Conv	32
Dense1	3
Dense2	4

Number of Bits Needed for Integer Parts of Weights

Weights	Number of Bits for Integer Part
conv_w	9
conv_b	1
dense_w	1
dense_b	1
dense_w2	2
dense_b2	3

1.2 C Model

The variables in software can be implemented in integers with a scaling factor of 256. When doing multiplication, the scaling factor of the result will accumulate to 256x256, so the result has to be divided by 256 or right shifted by 8 bits.

The non-linear functions of sigmoid and softmax don’t take up the major part of the computation. However, the exponential function needed in these functions is not very easy to implement in hardware. The sigmoid function is implemented in a polynomial approximation mentioned in the later section, and the absence of softmax

function theoretically doesn't hurt accuracy because of its monotonicity, though we observed an acceptable minor drop in accuracy possibly caused by numerical errors. The inference result will be directly given by calculating the argmax of the dense layer 2 vector.

As shown in the following table, the bottleneck of computation would first be the convolution layer and then the dense layer 1 since they are the most computation-hungary layers. They are the main focuses of our hardware accelerator.

The Proportional Running Time of Each Layer

Layer	Time
Conv	65.9%
Max Pooling	1.8%
Dense 1	31.9%
Dense 2	0.3%
Softmax	0.1%

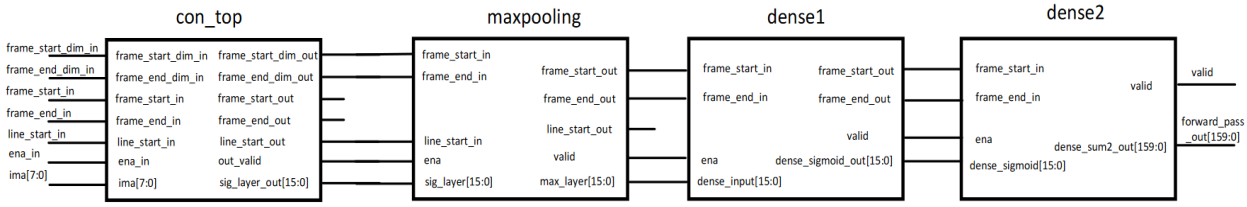
The data footprint has a 74% drop from 1069160 Bytes to 278952 Bytes since most 64-bit-wide double-precision floating-point variables are reduced to 16-bit-wide fixed-point variables.

One limitation of the C model is that it cannot really simulate timing, so we are leveraging another verification methodology that better handles it. We will introduce the verification methodology in Section 2.3.

Hardware Implementation

2.1 Hardware Architecture

The system diagram of hardware architecture is shown in the following figure.



The hardware CNN design is fully pipelined among all four sub-modules. In our system design, the ultimate goal is to improve frequency and reduce hardware resources used for processing one image.

The highest frequency a digital system can work on depends on the critical timing path. Pipelining the combinational logic in sub-modules (i.e. multipliers), the critical timing path in combinational logic is reduced and this system can operate at a higher frequency.

Within our hardware design, we tried to minimize the hardware resources for processing one image instead of parallelizing it to a higher extent. In fact, if the FPGA board that we implement our design has enough resources, we can obtain more parallelism simply by instantiating several blocks of the top module. Thus, we are choosing a fully pipelined architecture, which can reduce image buffers between sub-modules and reduce the time for which a sub-module waits. As we minimize the hardware usage, we can process more images in parallel and reduce execution time.

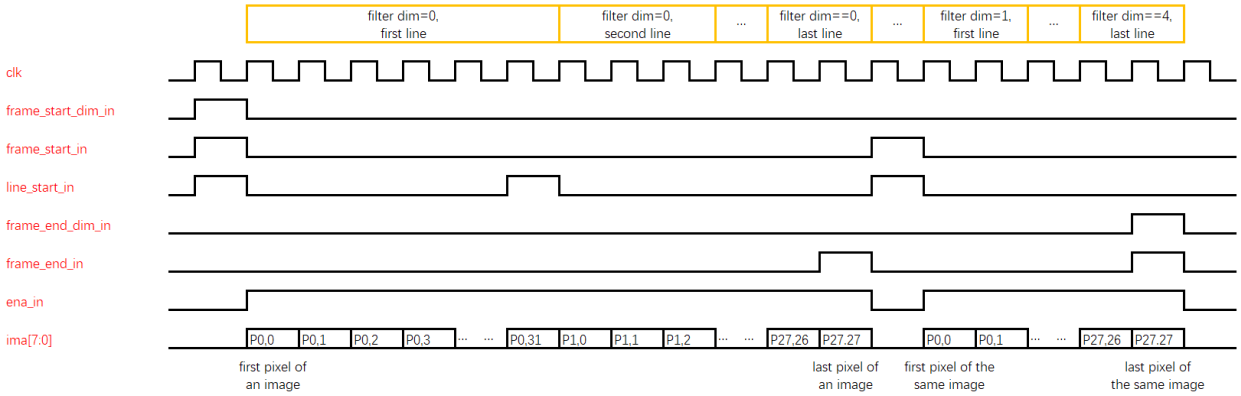
2.2 Block Diagram & Timing Diagram

(1) Top Module of Forward Pass

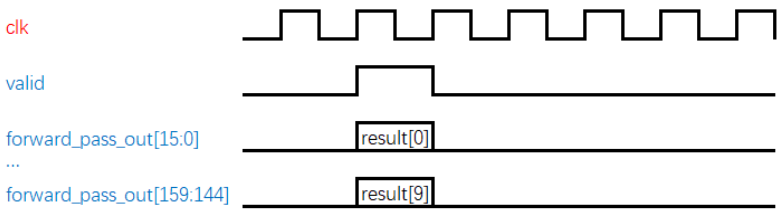
The input signals of the top-level module forward_pass.v is listed in the timing diagram below.

The signal *frame_start_dim_in* is a pulse that shows the start of a filter dimension, while *frame_end_dim_in* shows the end. The signal *frame_start_in* is a pulse that shows the start of an image, while *frame_end_in* shows the end of an image. As each image contains 5 filter_dims in the algorithm, the 5 images sent between the signal *frame_start_dim_in* and *frame_end_dim_in* should be exactly the same.

The *line_start_in* pulse indicates the start of a line of pixels and the enable signal shows when the image input *ima* is valid. The image input is sent to the top module in serial, one pixel after another.



Timing diagram of forward_pass_top input signals

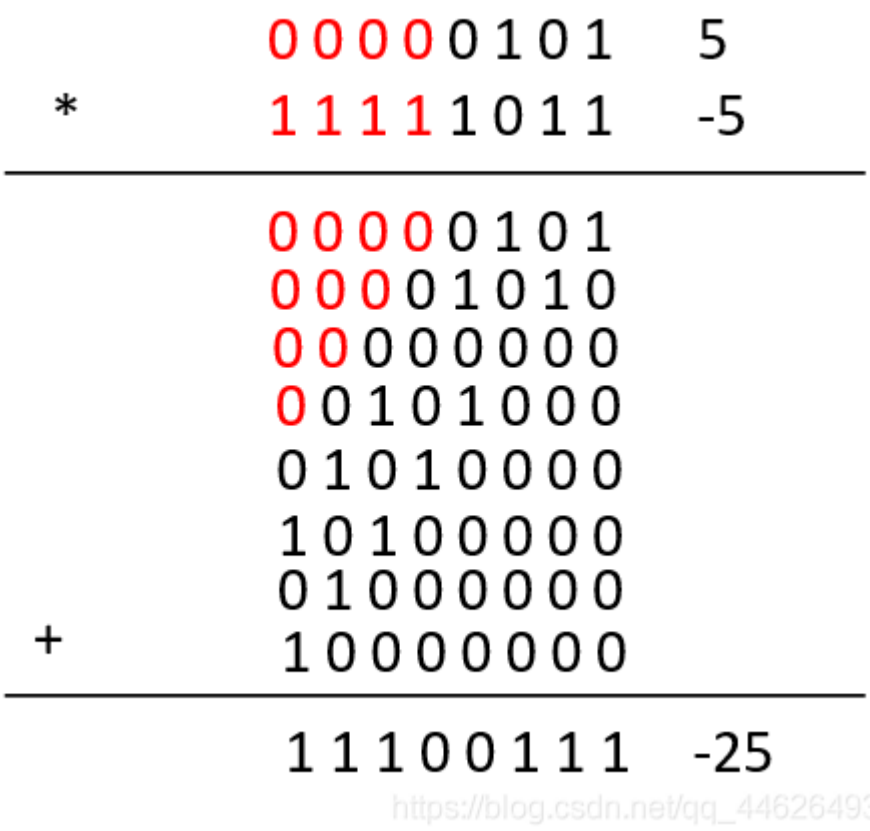


Timing diagram of forward_pass_top output signals

The output signal *forward_pass_out* is the predicted result for ten different digits when the *valid* signal is set to high level.

(2) Fixed Point Multiplication

The fixed point multiplication is made by the algorithm[1] below:



Extending the sign bit, make the width be double of the original input data width. For instance:

Input 16 bits 2's complement, then the width without sign bit is 15 bits. The maximum result width of the multiplication is 30 bits. With the necessary sign bit, the final output is 31 bits.

In this case, the input should be extended to 30 bits by copying the sign bit, then do the multiplication in the picture.

This multiplication used 3 clocks to get the result and is pipelined, extremely decreasing the timing requirement for setup time.

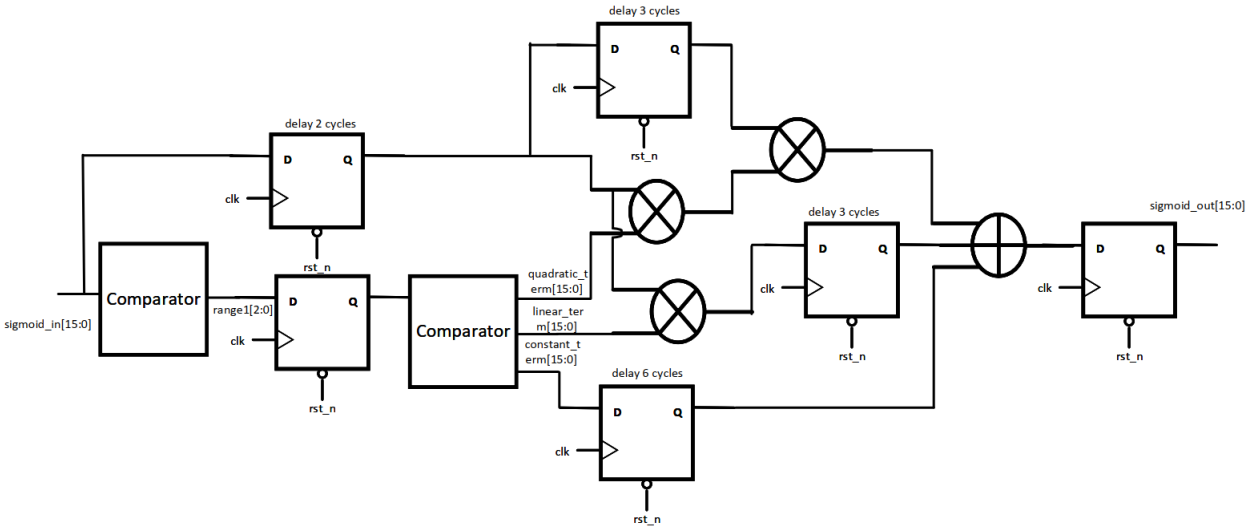
(3) Sigmoid

The sigmoid function of different intervals can be approximated using the following equations. [2] Considering that the 1 signed bit, 8 integer bits, 8 fractional bits fixed point number is not enough to correctly represent the coefficients with little error and that the return value of sigmoid is less than 1, the coefficients here can be represented using 1 signed bit, 0 integer bit, 15 fractional bits fixed point

number with more precision and no chance of overflow. The block diagram of sigmoid is shown below. The algorithm of our sigmoid function includes locating the input range, getting the quadratic, linear and constant terms, multiplication and accumulation.

Interval	Function Form
(1,1)	$y = 0.2383x + 0.5000$
[-2,-1]	$y = 0.0467x^2 + 0.2896x + 0.5118$
[-3,-2)	$y = 0.0298x^2 + 0.2202x + 0.4400$
[-4,-3)	$y = 0.0135x^2 + 0.1239x + 0.2969$
[-5,-4)	$y = 0.0054x^2 + 0.0597x + 0.1703$
[-5.03,-5)	$y = 0.0066$
[-5.2,-5.03)	$y = 0.0060$
[-5.41,-5.2)	$y = 0.0050$
[-5.66,-5.41)	$y = 0.0400$
[-6,-5.66)	$y = 0.0030$
[-6.53,-6)	$y = 0.0020$
[-7.6,-6.53)	$y = 0.0010$
[-∞,-7.6)	$y = 0$
[1,2)	$y = -0.0467x^2 + 0.2896x + 0.4882$
[2,3)	$y = -0.0298x^2 + 0.2202x + 0.5600$
[3,4)	$y = -0.0135x^2 + 0.1239x + 0.7030$
[4,5)	$y = -0.0054x^2 + 0.0597x + 0.8297$
[5,5.0218)	$y = 0.9930$
[5.0218,5.1890)	$y = 0.9940$
[5.1890,5.3890)	$y = 0.9950$
[5.3890, 5.6380)	$y = 0.9960$
[5.6380,5.9700)	$y = 0.9970$
[5.9700,6.4700)	$y = 0.9980$
[6.4700,7.5500)	$y = 0.9990$
[7.5500, +∞)	$y = 1$

Sigmoid function approximation[2]



Sigmoid function block diagram

(4) Convolution Layer

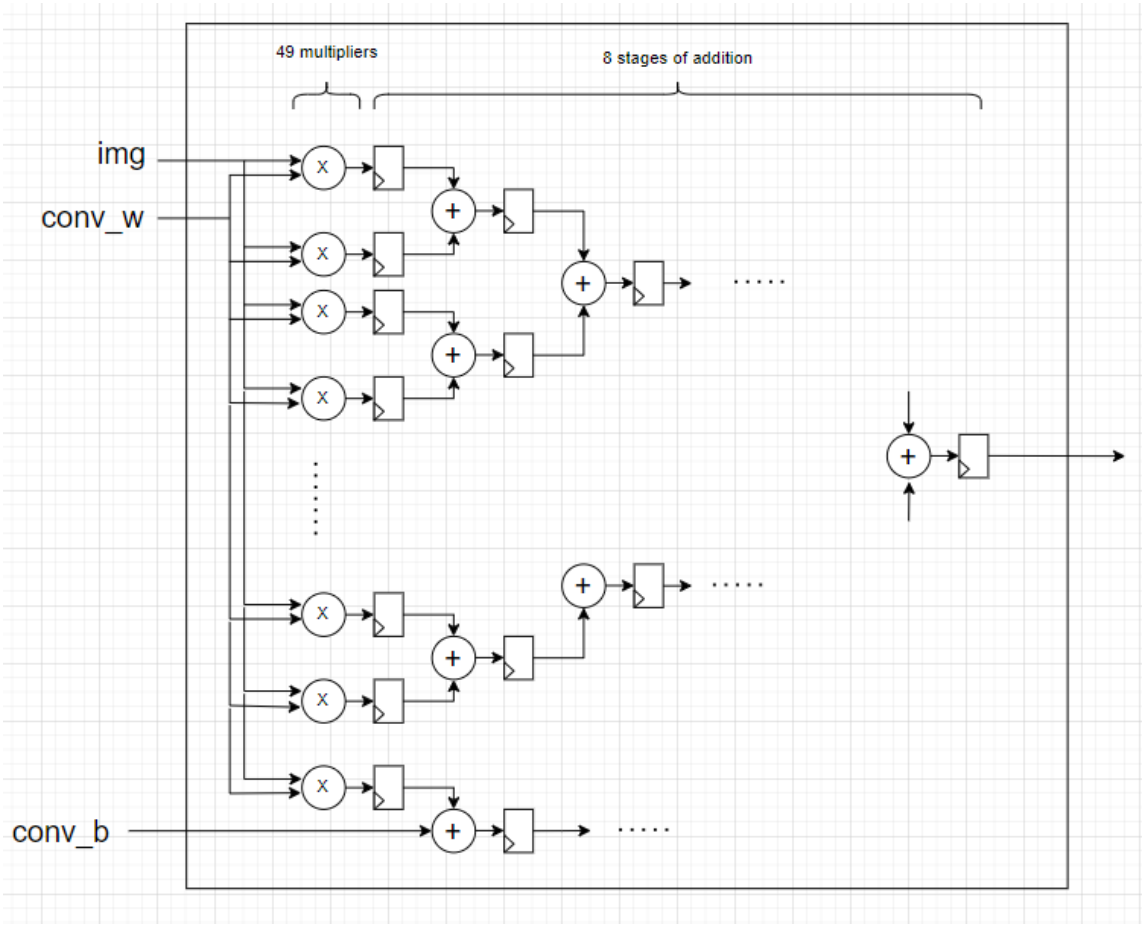
While one pixel is sent to the top level of the convolution layer each cycle, 49 pixels from 7 different lines are needed to perform the convolution calculation. Therefore, a SRAM storing 7 lines of pixel data is needed in our design. As shown in the *SRAM diagram of con_top*, the red block is the one pixel writing to SRAM and the yellow block is the 49 pixels read out for convolution calculation.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	7,0	7,1	7,2	7,3	7,4	7,5	0,6	0,7	0,8	0,9	0,10	0,11	0,12	0,13	0,14	0,15	0,16	0,17	0,18	0,19	0,20	0,21	0,22	0,23	0,24	0,25	0,26	0,27	0,28	0,29	0,30	0,31
1	1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	1,10	1,11	1,12	1,13	1,14	1,15	1,16	1,17	1,18	1,19	1,20	1,21	1,22	1,23	1,24	1,25	1,26	1,27	1,28	1,29	1,30	1,31
2	2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9	2,10	2,11	2,12	2,13	2,14	2,15	2,16	2,17	2,18	2,19	2,20	2,21	2,22	2,23	2,24	2,25	2,26	2,27	2,28	2,29	2,30	2,31
3	3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9	3,10	3,11	3,12	3,13	3,14	3,15	3,16	3,17	3,18	3,19	3,20	3,21	3,22	3,23	3,24	3,25	3,26	3,27	3,28	3,29	3,30	3,31
4	4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8	4,9	4,10	4,11	4,12	4,13	4,14	4,15	4,16	4,17	4,18	4,19	4,20	4,21	4,22	4,23	4,24	4,25	4,26	4,27	4,28	4,29	4,30	4,31
5	5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8	5,9	5,10	5,11	5,12	5,13	5,14	5,15	5,16	5,17	5,18	5,19	5,20	5,21	5,22	5,23	5,24	5,25	5,26	5,27	5,28	5,29	5,30	5,31
6	6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7	6,8	6,9	6,10	6,11	6,12	6,13	6,14	6,15	6,16	6,17	6,18	6,19	6,20	6,21	6,22	6,23	6,24	6,25	6,26	6,27	6,28	6,29	6,30	6,31
7	7,0	7,1	7,2	7,3	7,4	7,5	0,6	0,7	0,8	0,9	0,10	0,11	0,12	0,13	0,14	0,15	0,16	0,17	0,18	0,19	0,20	0,21	0,22	0,23	0,24	0,25	0,26	0,27	0,28	0,29	0,30	0,31
8	1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	1,10	1,11	1,12	1,13	1,14	1,15	1,16	1,17	1,18	1,19	1,20	1,21	1,22	1,23	1,24	1,25	1,26	1,27	1,28	1,29	1,30	1,31
9	2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9	2,10	2,11	2,12	2,13	2,14	2,15	2,16	2,17	2,18	2,19	2,20	2,21	2,22	2,23	2,24	2,25	2,26	2,27	2,28	2,29	2,30	2,31
10	3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9	3,10	3,11	3,12	3,13	3,14	3,15	3,16	3,17	3,18	3,19	3,20	3,21	3,22	3,23	3,24	3,25	3,26	3,27	3,28	3,29	3,30	3,31
11	4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8	4,9	4,10	4,11	4,12	4,13	4,14	4,15	4,16	4,17	4,18	4,19	4,20	4,21	4,22	4,23	4,24	4,25	4,26	4,27	4,28	4,29	4,30	4,31
12	5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8	5,9	5,10	5,11	5,12	5,13	5,14	5,15	5,16	5,17	5,18	5,19	5,20	5,21	5,22	5,23	5,24	5,25	5,26	5,27	5,28	5,29	5,30	5,31
13	6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7	6,8	6,9	6,10	6,11	6,12	6,13	6,14	6,15	6,16	6,17	6,18	6,19	6,20	6,21	6,22	6,23	6,24	6,25	6,26	6,27	6,28	6,29	6,30	6,31

SRAM of conv_top (first cycle)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	7,0	7,1	7,2	7,3	7,4	7,5	7,6	0,7	0,8	0,9	0,10	0,11	0,12	0,13	0,14	0,15	0,16	0,17	0,18	0,19	0,20	0,21	0,22	0,23	0,24	0,25	0,26	0,27	0,28	0,29	0,30	0,31
1	1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	1,10	1,11	1,12	1,13	1,14	1,15	1,16	1,17	1,18	1,19	1,20	1,21	1,22	1,23	1,24	1,25	1,26	1,27	1,28	1,29	1,30	1,31
2	2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9	2,10	2,11	2,12	2,13	2,14	2,15	2,16	2,17	2,18	2,19	2,20	2,21	2,22	2,23	2,24	2,25	2,26	2,27	2,28	2,29	2,30	2,31
3	3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9	3,10	3,11	3,12	3,13	3,14	3,15	3,16	3,17	3,18	3,19	3,20	3,21	3,22	3,23	3,24	3,25	3,26	3,27	3,28	3,29	3,30	3,31
4	4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8	4,9	4,10	4,11	4,12	4,13	4,14	4,15	4,16	4,17	4,18	4,19	4,20	4,21	4,22	4,23	4,24	4,25	4,26	4,27	4,28	4,29	4,30	4,31
5	5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8	5,9	5,10	5,11	5,12	5,13	5,14	5,15	5,16	5,17	5,18	5,19	5,20	5,21	5,22	5,23	5,24	5,25	5,26	5,27	5,28	5,29	5,30	5,31
6	6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7	6,8	6,9	6,10	6,11	6,12	6,13	6,14	6,15	6,16	6,17	6,18	6,19	6,20	6,21	6,22	6,23	6,24	6,25	6,26	6,27	6,28	6,29	6,30	6,31
7	7,0	7,1	7,2	7,3	7,4	7,5	7,6	0,7	0,8	0,9	0,10	0,11	0,12	0,13	0,14	0,15	0,16	0,17	0,18	0,19	0,20	0,21	0,22	0,23	0,24	0,25	0,26	0,27	0,28	0,29	0,30	0,31
8	1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	1,10	1,11	1,12	1,13	1,14	1,15	1,16	1,17	1,18	1,19	1,20	1,21	1,22	1,23	1,24	1,25	1,26	1,27	1,28	1,29	1,30	1,31
9	2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9	2,10	2,11	2,12	2,13	2,14	2,15	2,16	2,17	2,18	2,19	2,20	2,21	2,22	2,23	2,24	2,25	2,26	2,27	2,28	2,29	2,30	2,31
10	3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9	3,10	3,11	3,12	3,13	3,14	3,15	3,16	3,17	3,18	3,19	3,20	3,21	3,22	3,23	3,24	3,25	3,26	3,27	3,28	3,29	3,30	3,31
11	4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8	4,9	4,10	4,11	4,12	4,13	4,14	4,15	4,16	4,17	4,18	4,19	4,20	4,21	4,22	4,23	4,24	4,25	4,26	4,27	4,28	4,29	4,30	4,31
12	5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8	5,9	5,10	5,11	5,12	5,13	5,14	5,15	5,16	5,17	5,18	5,19	5,20	5,21	5,22	5,23	5,24	5,25	5,26	5,27	5,28	5,29	5,30	5,31
13	6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7	6,8	6,9	6,10	6,11	6,12	6,13	6,14	6,15	6,16	6,17	6,18	6,19	6,20	6,21	6,22	6,23	6,24	6,25	6,26	6,27	6,28	6,29	6,30	6,31

SRAM of conv_top (second cycle)



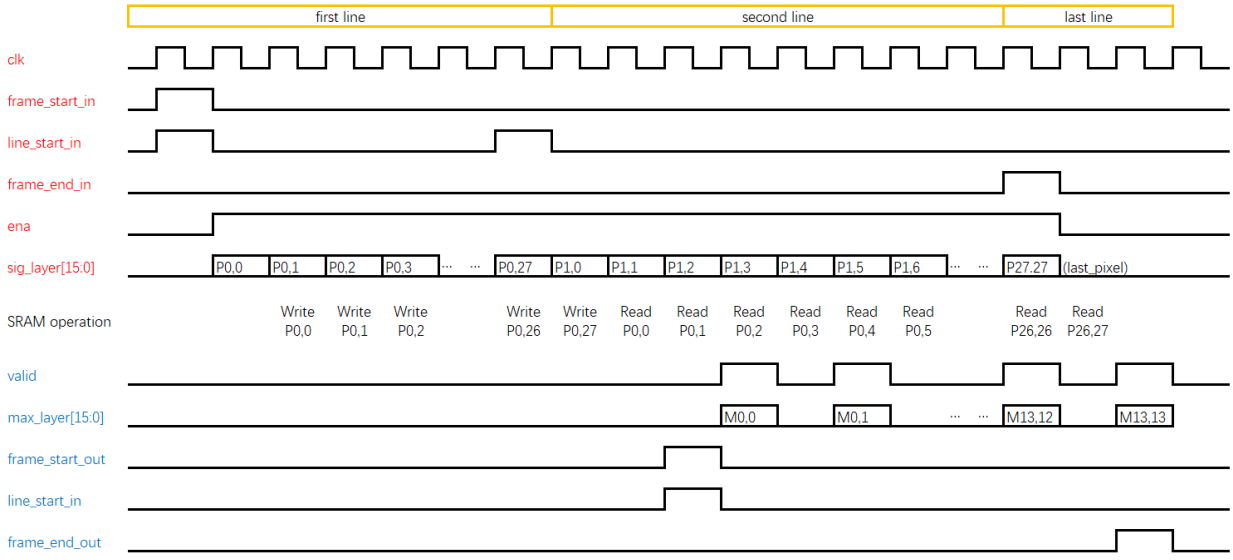
Block Diagram of Convolution Core

The conv layer is made by first multiplication, and gets 33 bits of data, then adds 2 of them in parallel each time.

For instance, after the multiplication, there are 49 data of 33 bits, add 48 of them, the 49th is added with bias. The result is 25 data of 34 bits, then do it again, but this time the last data is added with 0. Besides, after each calculation, there are registers to store the results, i.e., pipeline.

(5) Maxpooling

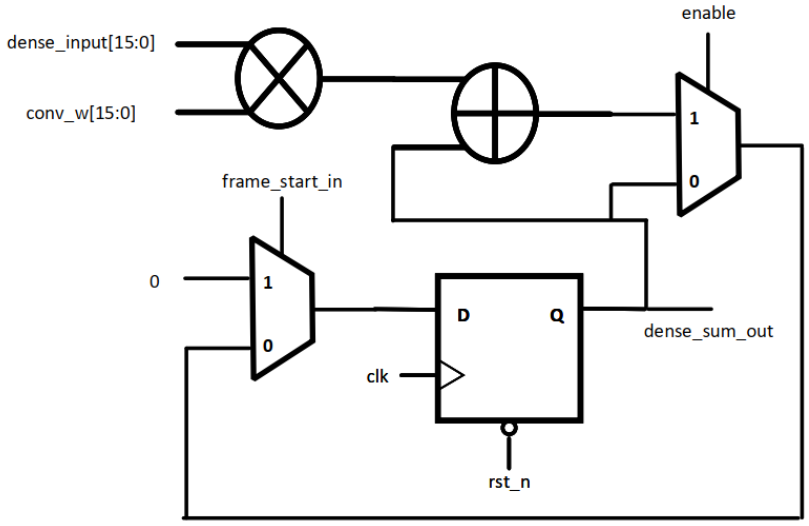
The maxpooling operation in this CNN algorithm is calculating the maximum value of a 2x2 block. In our hardware design, a SRAM is applied to store the pixels of line 0 and these pixels are read out when this maxpooling module is processing the pixels of line 1. Finally, the maxpooling result is the maximum value among two input pixels and two pixels read from SRAM.



Timing diagram of Maxpooling

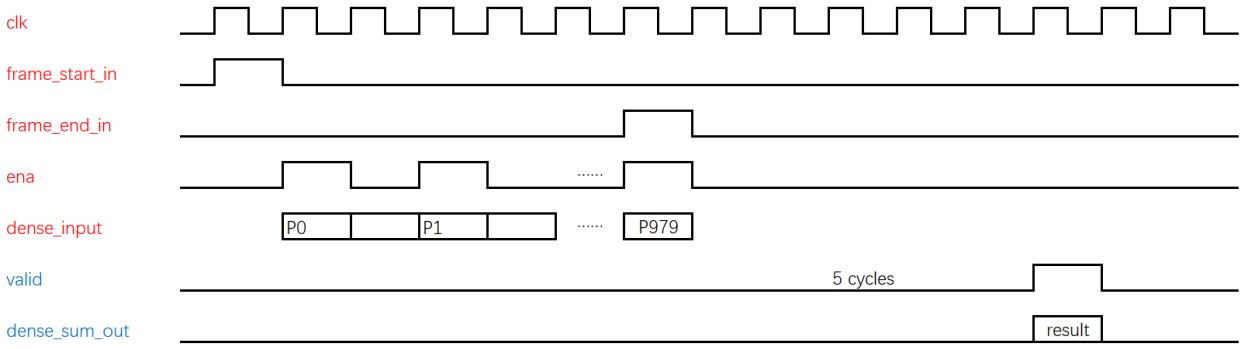
(6) Dense Layer

Multiply-accumulate is the main arithmetic operation in dense layer 1 and dense layer 2. It is only activated when the enable signal sent from the previous stage is triggered and the accumulator is cleared when the frame_start signal indicates the start of an input image.

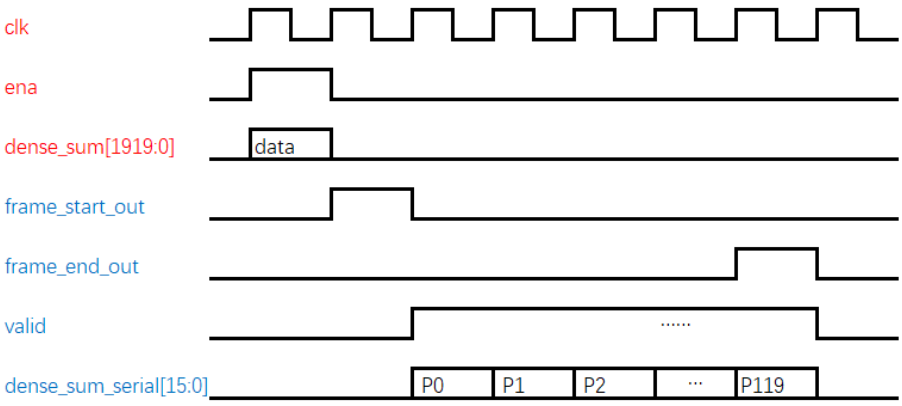


Multiply accumulator block diagram

For dense 1, 120 multiply-accumulators execute in parallel while each multiply-accumulator accumulates 980 times per image. For dense 2, 10 multiply-accumulators execute in parallel while each of them accumulates 120 times per image. As the dense_sigmoid results from dense 1 are in parallel, a parallel-serial converter is needed to fit the input of dense 2.



Timing diagram of Multiply-accumulator of Dense 1



parallel-serial converter between Dense 1 and Dense 2

2.3 Testbench and Verification Method

The testbench for each individual module mainly consists of five parts, including driver, monitor, transactions, reference model and scoreboard. The driver is a task that drives signals to the input ports of the design, while the monitor is a task that collects output signals from the design. The input transaction sent to DUT (Design Under Test) by the driver includes both special cases and random cases. Output transaction is then collected by the monitor.

The reference model is the golden vector we designed using C functions. Here, the SystemVerilog Direct Programming Interface (DPI) is used to import C functions to SystemVerilog. It is worth mentioning that the variables in golden vectors have a scaling factor of 256 and all rounding operations are rounded down, in order to fit the hardware design. Calling these functions in testbenches, we can have an expected output of the DUT. Finally, the scoreboard in testbench compares the expected output and actual output, showing the correctness of our design. A test case is passed when the results from C Model and DUT are exactly the same.

```c
#include <svdpi.h>
#include <math.h>
#include <stdio.h>
void dense_2_model(const int dense_w2[120][10], const int dense_sigmoid[120], const int dense_b2[10], int dense_sum2[10])
{
    // Dense Layer 2
    int i,j;
    for (i = 0; i < 10; i++)
    {
        dense_sum2[i] = 0;
        for (j = 0; j < 120; j++)
        {
                dense_sum2[i] += dense_w2[j][i] * dense_sigmoid[j];
        }
        //rounding <1,7,8>
        dense_sum2[i] = floor((double)dense_sum2[i] / 256);
        dense_sum2[i] += dense_b2[i];
    }

}
```

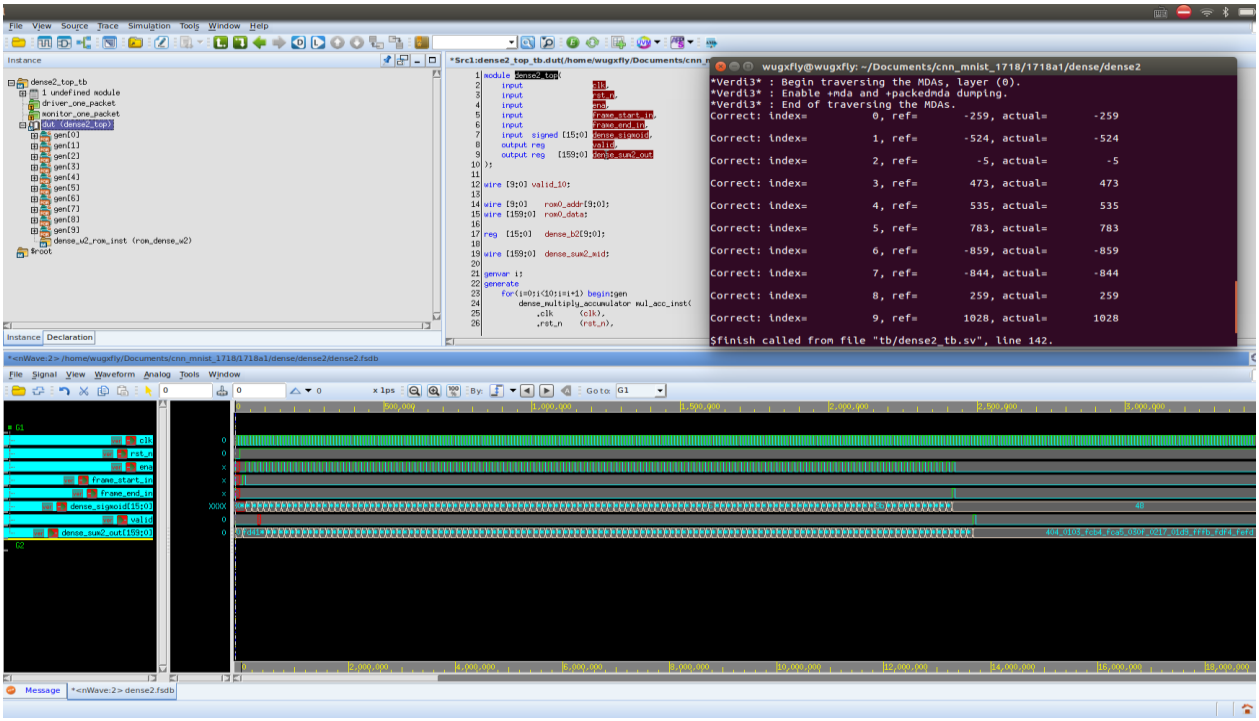
Golden Vector (C model) of Dense 2

```
import "DPI-C" function void dense_2_model(input int signed dense_w2[120][10], input int signed dense_sigmoid[120],
input int signed dense_b2[10], output int signed dense_sum2[10]);

 //reference model
 dense_2_model(
     .dense_w2 (dense_w2),
     .dense_sigmoid(tr_i.dense_sigmoid),
     .dense_b2 (dense_b2),
     .dense_sum2(dense_sum2_ref)
 );

 //scoreboard
 for(int u=0;u<10;u=u+1) begin
     if(dense_sum2_ref[u] != tr_o.dense_sum2[u])
         $error("Error: index=%d, ref=%d, actual=%d\n",u, dense_sum2_ref[u], tr_o.dense_sum2[u]);
     else
         $display("Correct: index=%d, ref=%d, actual=%d\n",u, dense_sum2_ref[u], tr_o.dense_sum2[u]);
 end
```

Calling the C model and comparing output results using testbench



An example of testing our design using testbench

2.4 HDL Synthesis

Resource	Utilization	Available	Utilization %
LUT	83228	2532960	3.29
LUTRAM	104	459360	0.02
FF	59559	5065920	1.18
BRAM	56.50	2520	2.24
IO	176	1456	12.09
BUFG	4	1440	0.28

Above is the resources needed for this design, about a million gates level.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.327 ns	Worst Hold Slack (WHS):	0.015 ns	Worst Pulse Width Slack (WPWS):	4.243 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	66618	Total Number of Endpoints:	66618	Total Number of Endpoints:	59737

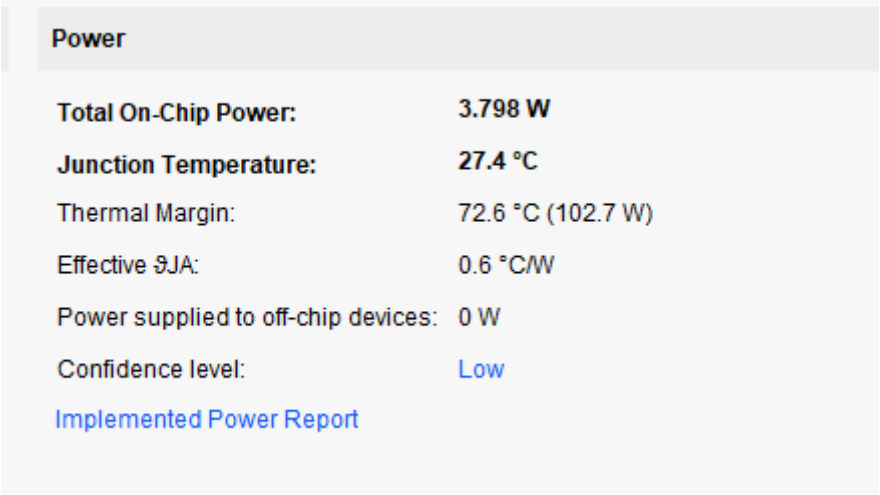
All user specified timing constraints are met.

And in the timing result, the Setup time is the bottleneck, hold time is well below constraints. Since this result is made under 10ns clock period/100MHz constraint, the theoretical maximum frequency is: 1 / 8.673ns = 115.3 MHz.

Therefore, it is guaranteed that the system can operate in 100 MHz with the timing requirements met.

Considering that our design only uses approximately 2% of the resources in the FPGA implementation, it is possible that we process several images at the same time to improve parallelism, which would further reduce execution time.

In order to minimize the hardware resources, the hardware design still needs improvement in dense layer 1. As the multiply-accumulators are enabled every 4 cycles (because of max pooling), we can reduce the number of multiply-accumulators from 120 to 30 by reusing them and controlling them with FSM.



The total power consumption of our design is 3.798W, which is much more power-efficient compared to CPU or GPU implementations whose typical TDPs are tens or hundreds of Watts, while our design out performs CPU or GPU implementation as shown below.

Compare with SIMD and openCL

Inference Time (across 600 images)

HW accelerator	34.8357ms (single unit) 4.354ms (8 units in parallel)
OpenCL	70.5105ms
SIMD	85.9332ms

Accuracy (across 600 images)

HW accelerator	0.888
OpenCL	0.907
SIMD	0.89

3.1 Comparison

The hardware implemented accelerator is approximately 50% faster than the OpenCL implementation, and 60% faster than the SIMD implementation. When considering the case where 8 HW accelerator units are put in parallel on a single FPGA, the time reduction becomes 93.8% against OpenCL and 94.9% against SIMD.

With the fixed point implementation in hardware, there is a minor loss in accuracy, which is approximately 1% to 2% compared to OpenCL and SIMD implementation.

3.2 Analysis

When using SIMD during the model inference, most arithmetic and data IO operations can be put into kernels and executed in parallel of 4. This significantly improves its performance upon the baseline implementation, both in forward and backward pass. When using a GPU combined with OpenCL kernels, the inference time of 600 images slightly improved to 70.5105ms, which is 18% faster than SIMD, while the GPU has much more compute units than the SIMD CPU. The potential bottleneck here is at the memory bus, where weights and biases of the model are sparsely stored in the CPU memory and need to be constantly fetched by the GPU.

Despite having the GPU cache, a single compute unit in the GPU might need to process data from different layers and nodes, which ignores the data reusability of weights and biases of the neural network. The excessive amount of data IO happens on the GPU also lowers the utilization of the compute units, which further reduces the actual performance from its expected performances.

For the hardware accelerator implemented in this assignment, there are two methods used to improve the overall performance: implementing on-chip ROMs to preload weights and biases; pipelining Arithmetic Logic Units(ALUs) to increase utilization.

Since the data bus between the HW accelerator and the CPU only supports 64-bit serial communication, it is impossible to transfer weights and biases together with image inputs during the inference time. On-chip ROMs are implemented according to the size and timing of each set of weights and biases, and will feed them to each ALU during the inference. This reduces the data IO, taking advantage of the neural network algorithm that: at each clock cycle, the input data required for each ALU is either determined (weights, biases), or ready to be used (outputs from previous ALUs).

Registers are used between ALUs (and some other combinational logics as well), to put the units in a pipelined order. With the correct adjustments mentioned in the previous section, all ALUs can reach almost 100% of utilization at all clock cycles, which is significantly better than what CPUs and GPUs can achieve.

Contribution of Each Team Member

Guoxian Wu: Hardware implementation, integration and Verification
Lingfeng Wu: Implement software model, help with hardware design
Haotian Zhu: Implement software model, help with hardware design
Zhuojun Yu: Hardware implementation of sub-module blocks

Reference

[1]https://www.allaboutcircuits.com/technical-articles/fixed-point-representation-the-q-format-and-addition-examples/

[2]T. H. Vu, R. Murakami, Y. Okuyama and A. Ben Abdallah, "Efficient Optimization and Hardware Acceleration of CNNs towards the Design of a Scalable Neuro inspired Architecture in Hardware," *2018 IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2018, pp. 326-332, doi: 10.1109/BigComp.2018.00055.