LUA FPGA-ACCELLERATED CNN

Execution of a Convolutional Neural Network forwarded on FPGA

Authors:
Ettore Randazzo
Benedetto Vitale
Professor:
Donatella Sciuto
Marco D. Santa
Course:
High Performance Processors and Systems

June 29, 2015

1 Introduction

The aim of this report is showing the procedure adopted in executing a CNN (Convolutional Neural Network) in Torch7(scientific computing tool for the programming language LUA). The following sections will show how a CNN, written in Lua, can be forwarded on a ZedBoard FPGA device, so that the CNN is executed on the FPGA, with an "ad hoc" programmed PL(Programmable Logic). In order to exchange data between the ZedBoard device and Torch, we are going to use FFI (Foreign Function Interface), so that we are able to write on the serial port to which the device is connected through the C programming language. Once the data is received and the computation is complete, the results are sent back to Torch by the ZedBoard, always through serial port, and read through FFI and C in Torch. In this paper we are going to show that the final result has been very similar to the one where the CNN was executed only in Torch.

2 Step I: from Torch to C

2.1 CNN's Torch implementation

First of all, we had to implement a working convolutional neural network in Torch.
We chose a very well known dataset, the USPS, which holds images of single digits as an input. Its goal is clearly to rightly classify the given image, guessing the digit number.

```lua
-- here we set up the architecture of the neural network
function create_network()
   local ann = nn.Sequential(); -- make a multi-layer structure
   -- 1x16x16
   ann:add(nn.SpatialConvolution(1,6,5,5)) -- becomes 6x12x12
   ann:add(nn.SpatialMaxPooling(2,2,2,2)) -- becomes 6x6x6
   ann:add(nn.Reshape(6*6*6)) --becomes 216
   ann:add(nn.Tanh())
   ann:add(nn.Linear(6*6*6,10)) --216 in, 10 out.
   ann:add(nn.LogSoftMax())

   return ann
end
```

This network is very simple:

- First, we receive an input consisting of one plane of 16x16 normalized doubles (which were the rgb numbers) and we create 6 output layers of 12x12 size, by using a frame window of 5x5.
- Second, we make a standar max pooling with stride 2 both on x and y axes and frame size of 2x2, thus reducing every plane's length to 6x6

- Third, we flatten the network, from 6x6x6 to 216 inputs to give to the classic feedforward neural network, where first we smoothed them by passing them through a tanh function
- Fourth, we connect the input layer with the output layer, composed of 10 neurons, each of them regarding one digit.
- Finally, for the sake of clarity we apply the LogSoftMax function to the output vector, which basically transforms it into 10 probabilities, which of course sum up to 1, and return the result.

As you may have realized, our network does not return a classification, instead it returns the probabilities of each output, which clearly contains the information to predict an output, by selecting the digit with the highest probability, but gives us also a very useful way to debug our net, as we can read every digit's digits prediction.

We trained the network with 10000 samples taken from a training set and tested it over 1000 test cases, achieving a 32/1000 error rate, which was better than results made by adding layers or changing the pooling, so we decided to stick with this simple, yet efficient network.

2.2 FFI: calling C in Lua

Next, we created a package called "fpgatorch" (for Torch), which contained the methods needed to forward some inputs, aswell as every input, in C, by using FFI. The package was very smooth: once installed, which needed to put it in the right directory, every time we opened Torch, we could have written require "fpgatorch", and we had everything we needed.

In order to invoke FFI, we needed to have a pre compiled library whence we put our only visible function: luaforward().

The first issue arose: by using FFI, we could have given simple data types such as chars or integers, but not Tensors (or arrays, or any structure). So we had two possibilities:

- Create a function with 256 input values
- Write down in a file the values, and read them from C

We clearly chose the latter, so we wrote the 256 doubles in a .bin file and read them in C.
Once we had the input in C, we needed to create a working convolutional neural network (synthesizable aswell), forward the input to it and get back the result. We did it step by step. First, we created the network and tried it in C, in order to see if it actually worked, by using double precision values (as in Torch), wrote the results down in a file, read them in Torch and evaluated them. As requested, we had to hard code the weights contained in the network, more

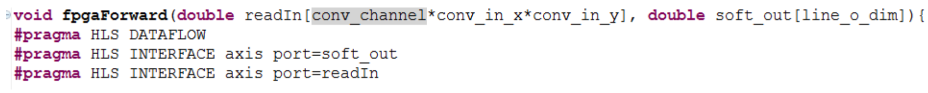
precisely the couples weight/bias of the convolutional layer and the feedforward neural network, by taking them from the trained network in Torch. They were several thousands which implied it was impossible to write them by hand, so we decided to write a script in Lua that taken a Tensor it would have returned its stringified version needed to initialize a C array with such weights, so we wrote these strings into some files, copied and pasted them in the C code. We also made this script open source. The result was a satisfying 32/1000 error rate: it was perfectly translated.

However, In order to synthesize it, we were recommended to use floats, because it appears that FPGAs don't treat well double precision floating point operations, so we casted to float the input received in C, and modified the network by using floats. Surprisingly, testing it in C gave the same previous error rate, so float resulted to be a costless gain.

2.3 From C to serial port

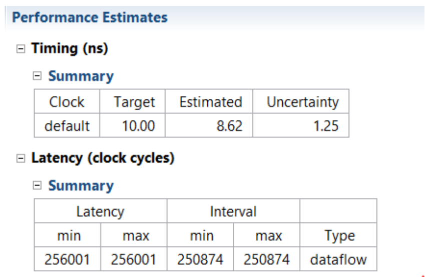
Once synthesized the net and programmed the FPGA, we had to send the input from C to the serial port where the FPGA was connected to, wait for an output to be received and forward it to Torch, in order to classify the net. Although it gave us some serious implementation issues, the concept is straightforward and does not need any further explanation.

In HLS, have been adopted the following PRAGMA's:



```
void fpgaForward(double readIn[conv_channel*conv_in_x*conv_in_y], double soft_out[line_o_dim]){
#pragma HLS DATAFLOW
#pragma HLS INTERFACE axis port=soft_out
#pragma HLS INTERFACE axis port=readIn
```

Input and Output has been synthesized as AXI-Stream interfaces, in order to map them respectively to the Read and Write port of the DMA(Direct Memory Access) that we have used in the block design to exchange data among the PL and the processing system. The Dataflow pragma has been inserted for performance reasons: it allows inner loops to begin execution as soon as the needed element of an array is ready, thus improving performance of almost 3000 clock cycles. The report related to the CNN HLS synthesis is the following:



Performance Estimates

⊟ **Timing (ns)**

⊟ **Summary**

Clock	Target	Estimated	Uncertainty
default	10.00	8.62	1.25

⊟ **Latency (clock cycles)**

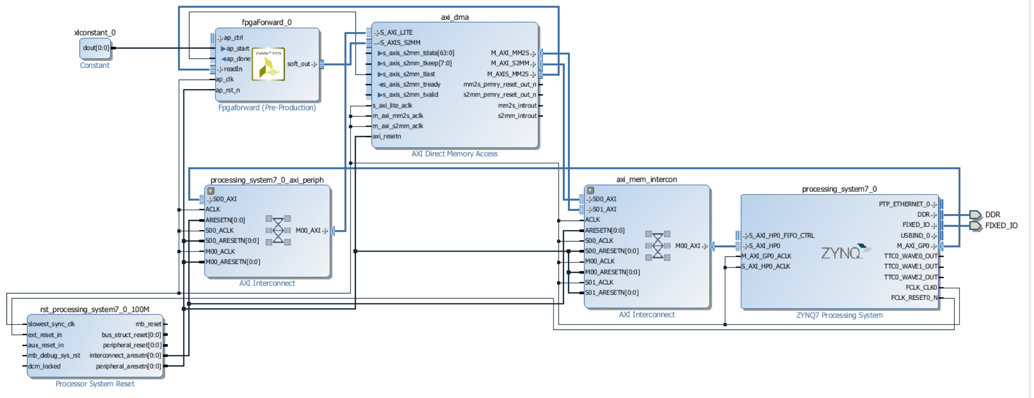
⊟ **Summary**

Latency		Interval		
min	max	min	max	Type
256001	256001	250874	250874	dataflow

As we can see, we have a total latency of 256001 clock cycles, with an estimated time of a little higher than 2 ms.

3 Step II: Vivado Block Design

Once the code has been synthesized and exported to an IP(Intellectual Property) block, we were ready to create a block design with the following structure:



Here are some considerations:

- The constant value 1 has been connected to the "ap_start" port of the "ap_ctrl" interface, since we want our block to be always available when data are transferred.
- To handle the transfers between the PL(Programmable Logic) and the Zynq Processing System we have used a DMA(Direct Memory Access) engine, made possible by the fact that "readIn" and "soft_out" have been synthesized as AXI-Stream interfaces, connected respectively to the "S_AXIS_S2MM" and "M_AXIS_MM2S" interfaces.

- The ”ap_done” signal of the same interface has been connected to the ”t_last” of the DMA, since the last packet is received when the function has complete its execution.
- The ”clock” and ”reset” signals of the ”fpgaForwrd” block has been connected to the ”Zynq Processing System” and to the ”rst_processing_system” respectively for synchronization issues.

4 Step III: SDK(Software Development Kit)

In SDK we have written a simple program that waits until data is not received and then stores them in a local SDK float array by calling the following function:

```c
//function to receive from UART port
void uart_receive(u8* buff,int RecvCount){
   int counter = 0;
   while(counter < RecvCount){
      while (!XUartPs_IsReceiveData(Config->BaseAddress));
         ;//waiting for input
      XUartPs_Recv(&uart,&buff[counter],1);
      counter++;
   }
}
```

The ”u8”(just one byte for each element)buffer is then then casted to float:

```c
//casting the buffer to float
in = (float *) buff;
```

The function uses the FPGA UART port to communicate with serial port(COM4), and so with C as shown before. The same things have been done for sending input back to the Serial Port after computation is complete:

```c
//function to send data to UART port
void uart_send(float* res_hw,int num){
   u8* buff = (u8*)res_hw;
   int counter = 0;
   while(counter < num){
      while(!XUartPs_IsTransmitEmpty(&uart));
         ;//waiting for other transfers to end
      XUartPs_Send(&uart,&buff[counter],1);
      counter++;
   }
}
```

The DMA has been used in simple mode trough the function provided by SDK:

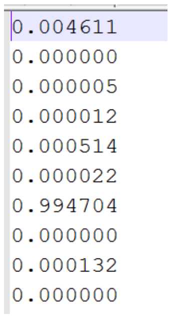
```
//Handling the DMA to PL input transfer
        do{
        status = XAxiDma_SimpleTransfer(&axiDma,(u32)in,sizeof(float) *
            (conv_channel*conv_in_x*conv_in_y),
                XAXIDMA_DMA_TO_DEVICE);
     }while(status != XST_SUCCESS);

        //waiting in order to make the transfer complete
     usleep(3000000);

      //receiving result from PL
     do{
        status =
            XAxiDma_SimpleTransfer(&axiDma,(u32)result,sizeof(float) *
            output_size,
             XAXIDMA_DEVICE_TO_DMA);
     }while(status != XST_SUCCESS);
```

After the data have been received, they are sent from the DMA to the PL by means of the simple transfer function, then, we wait for a time interval such that the DMA completes the transfer and the PL perform the prediction(we have tried also with "AxiDma_isBusy" check of SDK, but the PL ended up in infinite loop); then, after the PL writes the results into the relative register of the DMA, we transfer them to a local variable in SDK with another simple transfer and, finally, we send them back to the Serial Port through the function "uart_send".
Once, started, the program is kept into a loop until all the input has been transmitted and its relative results received. In order to make more than just one transfer we had to reset the DMA at each iteration through this instruction:

```
//resetting the DMA in order to transfer again
XAxiDma_Reset(&axiDma);
```

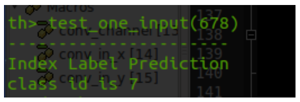
5 First Successful Test

After the final step in SDK, we have finally been able to perform some complete test, with a specific input(number 679/1000) of which we already knew the output of the CNN, which is the following:



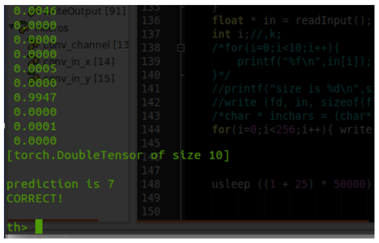
```
0.004611
0.000000
0.000005
0.000012
0.000514
0.000022
0.994704
0.000000
0.000132
0.000000
```

Then, we have followed all the iter explained in the precedent sections with the following command in Torch:



```
th> test_one_input(678)
------------------------
Index Label Prediction
class id is 7
```

And obtained the following result on the FPGA:

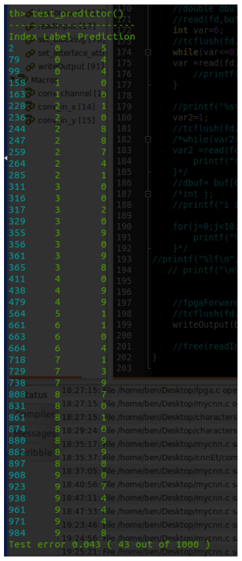


```
 0.0046
 0.0000
 0.0000
 0.0000
 0.0005
 0.0000
 0.9947
 0.0000
 0.0001
 0.0000
[torch.DoubleTensor of size 10]

prediction is 7
CORRECT!

th>
```

As we can see, the results are exactly the same.

6 Final Result and Conclusions

To conclude, we have made a test on the entire dataset, in order to perform a complete comparison between the results in Torch and on FPGA; the result have been the following:



```
th> test_predictor()
---------------------------
Index Label Prediction
2       0       5
79      0       4
99      0       4
158     1       0
163     1       0
228     2       1
236     2       0
244     2       8
247     2       8
259     2       7
264     2       4
285     2       1
311     3       0
316     3       0
317     3       2
329     3       0
355     3       9
356     3       0
361     3       9
365     3       8
411     4       0
438     4       9
479     4       9
564     5       1
661     6       1
663     6       0
664     6       4
718     7       1
729     7       3
738     7       9
808     8       7
831     8       0
861     8       1
874     8       0
880     8       9
882     8       9
897     8       0
908     9       0
923     9       7
938     9       4
961     9       4
971     9       4
984     9       8
Test error 0.043 ( 43 out of 1000 )
```

As the image above shows, we had 43 total mispredictions, against the 32 obtained by executing the net only in Torch, this may be due to the following reasons:

- Some values could have saturated at a certain point, thus loosing precision.
- The FPGA implementation loses accuracy in handling the float datatype, since, even if Torch uses doubles as default precision type for CNN, the net written in float has given the same accuracy of the one in double during the test performed by calling the software version of the net in C.