Reconfigurable Computing Lab

Philip Leong (philip.leongsydney.edu.au)
The University of Sydney

September 13, 2017

The long short-term memory (LSTM) network has been used with great success in time-series prediction problems. In this series of tutorials, we will develop an FPGA implementation of an LSTM.

Our implementation will have identical functionality to BasicLSTMCell[1] which is used in Google's Tensorflow neural network package. This, in turn, is an implementation of the network published in reference [2].

Using the same notation as [2], let $h_t^l \in \mathbb{R}^{n_l}$ be a hidden state in layer l at timestep t, $T_{m,n} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ be an affine transformation ($Wx + b$ for some $W \in \mathbb{R}$ and b), $\odot$ elementwise multiplication, sigm is the elementwise sigmoid function, tanh is the elementwise hyperbolic tangent, and h_t^0 be an input vector at timestep t. We use the activations h_t^L to predict y_t where L is the number of layers in the LSTM.

The neural network implements a state transition

$$LSTM : h_t^{l-1}, h_{t-1}^l, c_{t-1}^l \rightarrow h_t^l, c_t^l \tag{1}$$

where

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \text{tanh} \end{pmatrix} = T_{(n_{l-1}+n_l),(4n_l)} \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix} \tag{2}$$

$$c_t^l = f \odot c_{t-1}^l + i \odot g \tag{3}$$

$$h_l^t = o \odot \tanh(c_t^l) \tag{4}$$

A VirtualBox VM will be supplied that has TensorFlow, Vivado, and Vivado HLS already installed. The source files required to complete this tutorial are available from `https://github.com/phwl/hlslstm`.

In these labs, marks will be awarded not only for correctness but also understandability and elegance of the solution. Your answers to the questions should be in the form of a simple report. For each question below, provide a listing of the changes to the original code. Where applicable, give the speedup of your optimised design compared with the baseline single precision design generated in the first tutorial.

[1] `https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/ops/rnn.py`

1 Lab 1 - LSTM Network (Familiarlisation and Testbench)

This tutorial has the following goals:

- Practise translating published mathematical descriptions into hardware designs
- Gain experience in using high level synthesis
- Gain experience in design optimisation.

1.1 (20%) Familiarisation with xsimple.py

Download the tutorial source files using the command

```
git clone git@github.com:phwl/hlslstm.git
```

You can check that everything is working by typing:

```
cd hlslstm/src
make test
```

You should see output similar to the below.

```
python xsimple.py >xsimple.out
g++ -p   -o simple simple.cpp -lm
grep pred xsimple.out > /tmp/xsimple.out
echo "testing simple ..."
testing simple ...
(./simple | grep pred | sdiff -w80 /tmp/xsimple.out -)
vprint(y_pred) -0.0055398695 -0.01926   vprint(y_pred) -0.0055398695 -0.01926
vprint(y_pred) -0.012691636 -0.027500   vprint(y_pred) -0.012691636 -0.027500
vprint(y_pred) -0.019167556 -0.028273   vprint(y_pred) -0.019167556 -0.028273
vprint(y_pred) -0.021765375 -0.026982 | vprint(y_pred) -0.021765375 -0.026983
rm -f /tmp/xsimple.out
```

The output is the result of an sdiff command which compares the y_pred output of a floating-point python implementation of the LSTM (on the left hand side) with the output of the gen.cpp C program (on the right hand side). If the lines of text are different (as in the case in the last line), a | appears between them. Note that the simple program prints a number of variables, but (make **test**) only compares the y_pred outputs. The initial weight values are random so each time the xsimple.py program generates simple, the outputs will be different.

Modify the program so it accepts vectors of length 4, produces output predictions of length 4, has batch size 1 and sequence length 12. Also change the source code so the testbench containing main() and the lstm() function are in separate files.

1.2 (40%) Testbench

Also modify the program so it has a main program (which we will call the *testbench*), and reports the largest absolute error between two outputs, as well as the average mean squared error (MSE), where

$$MSE = \sqrt{(\sum_i (x_i - y_i)^2)}. \tag{5}$$

Also modify main() to return 1 in the case of an error and 0 if no errors occur. Although not a very satisfactory metric, we will call a MSE which is greater than 0.01 an error.

The expected output is provided in the l_y[] array defined in gen_io.h. Change the data type used for calculation from double to float. Rerun, is the MSE the same?

2 Lab 2 - LSTM Network (Parallelism)

Compiler directives are used in Vivado HLS to generate solutions with different architectures and degrees of parallelism. This tutorial has the following goals:

- Gain experience in optimising a design with compiler directives.

2.1 (50%) Vivado HLS Project

Create a Vivado HLS project [1] targetting the Zedboard (which has a xc7z020clg484-1 FPGA). Import the single precision version of your simple.cpp program.

You will need to modify the program as some parts of the original C program are not synthesisable, e.g. vprint(). In many cases, the original source code can be modified to support both normal C and HLS. In other cases, use C preprocessor *#ifdef* constructs so you can invoke both the synthesisable and original programs from the same source code.

Generate a synthesis report for the project, with a target clock period of $10ns$. How many cycles does the execution take? What is the actual clock period?

2.2 (50%) Compiler Directives

Explore how compiler directives such as DATAFLOW, ARRAY PARTITION and PIPELINE can improve the speed of the design. Use the Analysis Perspective to understand the timing implications of your changes. Find configurations which give the best overall result in terms of: number of cycles T; area A (defined as the percentage of the most used resource out of LUTs, memories and DSPs); and AT product.

3 Lab 3 - LSTM Network (Precision)

Fixed-point arithmetic requires less area and has lower latency than floating-point. Moreover, FPGAs can more efficiently perform low-precision calculations than processors and graphics processing units. This tutorial has the following goals:

- Gain experience in converting floating-point designs to fixed-point.
- Quantify the performance advantages of fixed-point over floating point for our long short-term memory (LSTM) example.

3.1 (50%) Fixed-point C Implementation

Use the Vivado HLS ap_fixed type to modify your floating-point implementation to use ap_fixed<12,4, AP_TRN,AP_WRAP>. You will also need to deal with the implementations of mytanh() and mysigmoid() as ap_fixed does not support them. Replace both functions with a lookup table.

The first two numbers represent the word length and number of integer bits respectively. What setting of the number of integer bits gives the lowest error?

3.2 (50%) RTL Simulation

Synthesise the design using HLS. Run the C simulation to verify the output is correct. Moreover, run a C/RTL cosimulation to check that the HLS output is correct.

4 Lab 4 - LSTM Network (Exploration)

In this lab we explore how to obtain the maximum speed in our LSTM implementation.

- Gain experience in exploring alternative implementations to accelerate LSTM.

For the questions in this lab, you can apply any optimisations as long as the the percentage error for any output (compared with the original double precision C version) is less than 10%.

4.1 (50%) Optimising Number of Cycles

In this lab, you are invited to modify the C source code, introduce approximations, trying different fixed point rounding modes, changing directives etc in order to achieve the minimum number of cycles for the LSTM implementation at 100 MHz.

4.2 (50%) Optimising Execution Time

Up to this point in all the labs, we have only endeavoured to optimise the number of cycles, T. In this part, optimise the execution time ($T \times p$) where p is the clock period, by setting tighter timing constraints.

5 Lab 5 - LSTM Network (Interface I)

In this two-part lab we create an interface to our accelerator for use by a CPU with the following goals:

- Gain experience making a host processor to FPGA interface.

5.1 (100%) Hardware Integration

Adapting the design in reference [1], create a Zynq design for the xc7z020clg484-1 FPGA on the Zedboard with your LSTM design included as an IP Block. In this lab you do not need to verify that it works, you just need to integrate the blocks.

6 Lab 6 - LSTM Network (Interface II)

This lab question is a continuation of the previous one.

6.1 (60%) Software-Hardware Co-design

In this lab, create a design which combines the Zynq processor on the Zedboard, and the LSTM hardware IP developed in the previous tutorial.

6.2 (40%) Optimisation

Compare the performance of the accelerated design with a pure Zynq implementation and the original desktop implementation. Improve the performance of the accelerated design using either software or hardware techniques.

7 Lab 7 - Energy Optimisation

In many embedded applications, energy consumption is an important design objective. This lab has the goals of:

- Understanding energy tradeoffs in our implementation.

7.1 (50%) Measurement

Compare the performance of the accelerated design with a pure Zynq implementation, the original desktop implementation and the accelerated implementation in terms of power and energy.

7.2 (50%) Optimisation

Optimise the design in order to obtain the lowest energy. This is best done by maximising the speed of the LSTM core as the power consumption of the FPGA does not change significantly with different designs.

References

[1] Xilinx. UG871 vivado design suite tutorial. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug871-vivado-high-level-synthesis-tutorial.pdf`.

[2] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014.