Reconfigurable Computing Lab

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

September 29, 2018

The long short-term memory (LSTM) network [6, 3] has revolutionised approaches to time-series prediction problems such as speech recognition and machine translation. In this series of tutorials, we will develop an FPGA implementation of an LSTM [7].

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 references [3, 10].

Using the same notation as [10], 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 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

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

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

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

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.

Only a small number of FPGA implementations of LSTM networks [6] have been reported. Chang et. al. [1] in 2015 implemented an LSTM network on a Zynq 7020 FPGA using a matrix-vector multiplier architecture. Their 2 layer, 128 hidden unit design operated at 142 MHz and was 21 faster than the ARM Cortex A9 processor on the same FPGA. In 2017 Guan et. al. [4] in 2017 describe an optimised 32-bit floating point LSTM implementation which achieved 7.26 GFLOPS on a Xilinx Virtex7-485t FPGA at 150MHz. Finally, Han et. al. [5] describe a load-balance-aware pruning method to introduce sparsity and quantisation in an LSTM implementation. On a Xilinx XCKU060 FPGA running at 200MHz with 12-bit precision, their design achieved 282 GOP/s, corresponding to 2.52 TOPS on an equivalent dense network.

[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 (40%) Familiarisation with xsimple.py

A VirtualBox virtual machine (VM) will be supplied that has TensorFlow, Vivado, and Vivado HLS already installed (the username and password are both "vivado"). Instructions on how this VM was created are given in Appendix A. After starting up the VM, download the tutorial source files and set up Tensorflow using the commands

```
git clone https://github.com/phwl/hlslstm
workon keras_tf
```

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 xsimple.py 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 file containing main() (which we will call the *testbench*) and the lstm() function are in separate files.

1.2 (60%) Testbench

Let h_i^p be the prediction of the LSTM (i refers to an element of h, and p is the input pattern number) and y_i^p is the double precision expected result ($l\_y[p][i]$ in gen_io.h). Let P be the number of patterns (L_PATS in gen.h). Modify the program so the testbench reports the largest relative error between two outputs

$$MAXERR = max_{i,p}|(h_i^p - y_i^p)| \quad \forall i, \forall p \tag{5}$$

Also print the average mean squared error (AMSE), where the mean square error (MSE) between two vectors of length N, h and y, is

$$MSE^p(h, y) = \frac{1}{N} \sum_{i=1}^{N} (h_i^p - y_i^p)^2. \tag{6}$$

and

$$AMSE = (\sum_{p=1}^{P} (MSE^p))/P. \tag{7}$$

Also modify main() to return 1 in the case of an error (and 0 otherwise). Although not an entirely satisfactory metric, we will call a AMSE which is greater than 0.01 an error.

Comment out the line in the Makefile that runs the xsimple.py program as it will overwrite your C source files. The expected output of the LSTM network is provided in the l_y[] array defined in gen_io.h. Change the data types used for the C program from double to float by modifying the header and cpp files. Rerun, is the AMSE 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 [9] targetting the Zedboard (which has a xc7z020clg484-1 FPGA). This should be done outside the Tensorflow environment. The way you can tell is that if you are in the Tensorflow environment, the command prompt begins with (keras_tf). Import the single precision version of your simple.cpp program.

You may 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 such 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 need to do the following

- Change references to floating point variables to ap_fixed<12,4,AP_TRN_ZERO,AP_SAT>. One way is to declare

  ```
  #define W_WIDTH 12
  #define I_WIDTH 4
  typedef ap_fixed<W_WIDTH,I_WIDTH,AP_TRN_ZERO,AP_SAT> l_t;
  ```

  in gen.h and then declare all fixed point variables as l_t. W_WIDTH is the word length and I_WIDTH is number of integer bits (the number of fractional bits is W_WIDTH−I_WIDTH).

- Add

  ```
  #include <ap_fixed.h>
  ```

  to the appropriate files and change CFLAGS in the Makefile to

  ```
  CFLAGS= -I/opt/Xilinx/Vivado_HLS/2015.4/include
  ```

  so the C compiler can find the include file.

- Fix the testbench so it can deal with floating point inputs and outputs but execute in fixed point.
- Deal with the implementations of mytanh() and mysigmoid() as ap_fixed does not support them. One solution is to replace both functions with a lookup table (if you do it this way, you should make it flexible so the number of integer and fractional bits can be changed). Another is to use math primitives supported by the ap_fixed type.

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)

In this 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 (20%) Hardware Integration

Adapting the design in reference [9], 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.

5.2 (40%) 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.

5.3 (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. Ideas include: finding better ways to compute the nonlinearities; reducing precision requirements; accumulating at a higher precision before rounding; and increasing the clock rate of the design.

6 Report

Report on the work done in these labs in IEEE conference format. Templates are available at `https://www.ieee.org/conferences_events/conferences/publishing/templates.html`. The maximum length of the report is 4 pages but you can include source code listings etc in additional appendices. In your report, you should include background on machine learning, LSTM, cite key papers etc. Examples of papers that you can use as templates include [3, 8, 2] (the last one is a 4 page paper).

A Installing Vivado, Python and Tensorflow

Here is the sequence of commands for installing Python and Tensorflow on an Ubuntu 16.04 system.

```
sudo apt-get install libboost-all-dev libusb-1.0-0-dev python-mako doxygen python-
    docutils cmake build-essential
sudo apt-get install python-pip
pip install --upgrade pip
sudo pip install virtualenv
sudo pip install virtualenvwrapper
export WORKON_HOME=~/Envs # needed later for virtualenvwrapper
mkdir -p $WORKON_HOME

source /usr/local/bin/virtualenvwrapper.sh
mkvirtualenv keras_tf
pip install tensorflow
pip install keras
pip install requests
```

You should also add the following to /.profile.

```
export WORKON_HOME=~/Envs # needed later for virtualenvwrapper
source /usr/local/bin/virtualenvwrapper.sh
```

Vivado and Vivado HLS can be installed according to the instructions from Xilinx. For Ubuntu 16.04 you must also put the following line in /etc/udev/rules.d/70-persistent-net.rules and reboot (this is needed to change the ethernet device from en0 to eth0 which is needed for the Xilinx license system)

```
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="02:01:02:03:04:05",
    ATTR{dev_id}=="0x0", ATTR{type}=="1", NAME="eth0"
```

References

[1] Andre Xian Ming Chang, Berin Martini, and Eugenio Culurciello. Recurrent neural networks hardware implementation on FPGA. *CoRR*, abs/1511.05552, 2015.

[2] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello. Hardware accelerated convolutional neural networks for synthetic vision systems. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 257–260, May 2010.

[3] A. Graves, A. r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, May 2013.

[4] Yijin Guan, Zhihang Yuan, Guangyu Sun, and Jason Cong. Fpga-based accelerator for long short-term memory recurrent neural networks. In *22nd Asia and South Pacific Design Automation Conference, ASP-DAC 2017, Chiba, Japan, January 16-19, 2017*, pages 629–634. IEEE, 2017.

[5] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 75–84, New York, NY, USA, 2017. ACM.

[6] Sepp Hochreiter and JÃijrgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[7] Siddhartha, Yee Hui Lee, Duncan J.M. Moss, Julian Faraone, Perry Blackmore, Daniel Salmond, David Boland, and Philip H.W. Leong. Long short-term memory for radio frequency spectral prediction and its real-time FPGA implementation. October 2018.

[8] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A framework for fast, scalable binarized neural network inference. In *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 65–74, 2017. Source code available from https://github.com/Xilinx/BNN-PYNQ.

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

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