Software Reference Model ©FABU America

Abinash Mohanty

March 2, 2019

Contents

1 Introduction

This document is an introduction to Software Reference Model (`SW RefModel`). `SW RefModel` is a `Python 2.7` based wrapper around `Tensorflow 1.3.1`. Its purpose is to create a easy to use framework which can have multiple network architectures defined, models and data stored and FABU DLA hardware blocks modeled. Framework organization is inspired from the open source implementation of Faster-RCNN from here. The framework internally uses Tensorflow to do major portion of the computations and then converts the results to a format similar to the hardware output.

It performs the following tasks:

- Provide golden output for validation framework to test RTL (high level modeling of the FABU hardware is necessary for this).
- Convert float32 models to 8bit models (this conversion can be done using utility scripts in lib/utils directory).
- Merge Batch normalization layers into convolution layers (DLA doesn't support BN).
- Maintain a clean dictionary structure storing layer information each network so that it can efficiently provide data to validation framework.
- Pixel Tracker (given a network, layer, channel number and pixel location, it can give out partial accumulation results) to assist in detailed debug.

Requirements:

- Python 2.7.15 (recommended: Anaconda])
- Packages like: python-opencv, easydict, matplotlib, cv2, numpy, etc...
- Tensorflow

The bird's eye view structure of `SW RefModel` is shown below. From here on we shall denote the root directory of `SW RefModel` as `$REF_ROOT`. Unless stated otherwise, from here on all directories will be addressed relative to `$REF_ROOT`. `refModel.py` is the entry point to `SW RefModel`. `lib` directory contains the core implementations. Details of individual modules and sub-modules will be provided in subsequent sections.

```
SW RefModel
│
├─ lib
│  │
│  ├─ networks (All networks and tf wrappers are defined here)
│  │  └─ network.py⋆ | factory.py | pvanet_8bit_ob.py⋆ ...
│  │
│  ├─ fp (fixed point to floating point conversion)
│  │  └─ fp_op.cc⋆ | fp.sp | fp_op.py ...
│  │
│  ├─ lp (right shift, truncation and saturation)
│  │  └─ lp_op.cc⋆ | lp_op.py | lp.so ...
│  │
│  ├─ nms (python software baseline for nms )
│  │  └─ cpu_nms.c | cpu_nms.pyx | cpu_nms.so | py_cpu_nms.py ...
│  │
│  ├─ roi_pooling_layer (Roi Pooling modeling)
│  │  └─ roi_pooling_op.cc⋆ | roi_pooling.so | roi_pooling_op.py ...
│  │
│  ├─ rpn_msr (Proposal layer modeling)
│  │  └─ proposal_layer_tf.py⋆ | sqrootLookUp.py | generate.py ...
│  │
│  ├─ saturate (saturation modeling)
│  │  └─ saturate_op.cc⋆ | sp.so | make_ref_hw.sh | saturate_op.py ...
│  │
│  ├─ utils (utility functions)
│  │  ├─ blob.py | parse_args.py | fixed_point.py | bbox_transform.py ...
│  │  │
│  │  └─ convert_model (Convert caffe model to tensorflow model)
│  │     └─ convert.py ...
│  │
│  └─ config.py⋆ (emulator configuration file)
│
├─ output (Tensor board dumps))
│  └─ events.out.tfevents.1541056279.decacs1 ...
│
├─ refModel.py⋆ (Entry Point to SW RefModel ! )
│
└─ README.md
```
[1]

[1]Files with ⋆ beside their name are important files the user should look at to understand the reference model better.

In the subsequent sections we will go through the entire framework in detail. Sections are partitioned such that there is minimum overlap between them. Users can directly jump to any section they are interested in.

2 Reference Model

`$REF_ROOT/refModel.py` is the primary entry point of `SW RefModel`.

2.1 Standalone Application Mode

When executed in standalone mode, it executes the given network end-to-end and displays[2] the output.

To run reference model in standalone mode use the following command: `python refModel.py --network pvanet_8bit_ob --hw_sim 1 --image 000.jpg --resolution_mode 0`

Description of the arguments is given below:

- network: type is string. Its the name of the network as implemented in `$REF_ROOT/lib/networks/factory.py`. Default: `pvanet_8bit_ob`
- hw_sim: flag to choose between hardware modeling or pure 32bit floating software implementation. Default: `1`
- image: filename of the image. it should contain the path relative to `/global/mobai2/cnn_emu_data/data`. Default: `000.jpg`
- resolution_mode: reference model supports 3 resolution modes. 0: high resolution, 1: mid resolution, 2: low resolution. The network class is responsible for implementing this in the pre-process method. Default: `0`

```python
if __name__ == '__main__':
  ...
  tf.reset_default_graph()
  sess = tf.Session(config=tf.ConfigProto(
   allow_soft_placement=True))

  # create network instance and load models parameters
  net = get_network(str(em_network), em_isHardware)
  sess.run(tf.global_variables_initializer())
  net.load(model,sess)

  # Execute the method for sw demo in network class
  net.run_sw_demo(sess, em_image, 0)
```

[2]provided that the network class of the given network as all the necessary implementations like pre/post process, visualize output etc.

2.2 Data for Validation Mode

Reference model provides data for validation or RTL. The `HWRefModel`. This is achieved through methods like `get_data_for_validation()` and `get_bias()`.

2.2.1 get_data_for_validation

```
def get_data_for_validation(net, em_image, scale_mode,
    em_start_layer, em_end_layer):
  """
  Efficiently get the inputs, outputs, weights of
   layers given by start and end index
  """
```

Description of arguments are given below:

- net: type is string. Its the name of the network as implemented in `$REF_ROOT/lib/networks/factory.py`.
- em_image: path to the input image (user is responsible to send in appropriate image (classification vs detection)). it should contain the path relative to `/global/mobai2/cnn_emu_data/data`.
- scale_mode: scaling mode (0: highest resolution, 1: mid resolution, 2: low resolution). The network class is responsible for implementing this in the pre-process method.
- em_start_layer: ID of the start layer (this should be same as implemented in `net_XXX._layer_map`).
- em_end_layer: ID of the end layer (this should be same as implemented in `net_XXX._layer_map`)

Description of return data structure from this method:

- List called multi_layer_outputs. Each element corresponds to a layers to be tested from em_start_layer to em_end_layer for each layer: [layer_output, layer_inputs, parameters] to access data for layer I, (x = I - em_start_layer):

– layer output:	multi_layer_outputs[x][0]
– layer inputs (j):	multi_layer_outputs[x][1][j]
– layer weights:	multi_layer_outputs[x][2][0]
– layer biases:	multi_layer_outputs[x][2][1]

- layer info dictionary (`netXXX._layer_map`)

The values return by this method are in float32 format. The consumer (`HW RefModel`) is responsible for converting to `int8` or `int16`. This method run the network to generate the inputs and outputs of layers. For the model parameters it parses the model file and gets weights and biases related to any particular layer (using the layer "name" from `XXX_net._layer_map`).

2.2.2 get_bias

`HW RefModel` needs all the bias values at the same time to initialize the bias ram prior to testing any particular layer. For this purpose `SW RefModel` has this API to give out all the bias values at the same time.

```
def get_bias(net):
  """
  API to return all the bias values in the model. To
   be used by validation framework to initialize the
   bias SRAM.
  Pre-req: The model file (netname.npy) must be
   present in cfg.MODELS_DIR
  Arguments:
    net: name of the network.
  Returns:
    list containing all the bias values for all the
   layers in same sequence as define in net._layer_map
    dict.
  """
```

The method takes in the name of the network as a string and using that looks for appropriate model file (`XXX.npy`). It then scans through the entire file, grabs all the bias values and packs them into a list and sends it out.

The length of the output list is equal to the number of layers in the given network. If for a layer has no bias value (ex. proposal layer, roi pooling etc) then that index of the list will have an empty list ([]). For all other layers, the output list will have a list of bias values (whose length is equal to the number of output channels (conv) or number of output nodes (fc)). The bias values returned are in `fixed point` format (`int8` or `int16`)

3 Networks

The wrapper around Tensorflow and all the networks are implemented in this module. All the layers supported are abstracted in `Network` class (`$REF_ROOT/lib/networks/network.py`). Its done so as to have clean network implementations and reuse most of the codes. Every supported network is implemented as a new class which inherits the base `Network` class. The base class implements all the methods that are common to all networks. Network specific methods, like `pre_process`, `post_process`, `setup` etc, are implemented in the child classes.

3.1 Network class

3.1.1 Loading model parameter

```
def load(self, data_path, session, ignore_missing=
 False):
  """
  Initialize graph with pre-trained parameters.
  """
  data_dict = np.load(data_path).item()
  for op_name in data_dict:
    with tf.variable_scope(op_name, reuse=True):
      for param_name, data in data_dict[op_name].
 iteritems():
        try:
          var = tf.get_variable(param_name)
          session.run(var.assign(data))
          print_msg("assign pretrain model "+
 param_name+ " to "+op_name,0)
        except ValueError:
          print_msg("ignore "+str(param_name),3)
          if not ignore_missing:
            raise
  print_msg("Model was successfully loaded from "+
 data_path ,3)
```

3.1.2 Convolution

For convolution, it is assumed that there is no overflow during accumulation in the hardware. SW Reference model uses Tensorflow's built-in function `tf.nn.conv2d()` to perform integer convolution first. On top of that the hardware is modeled using a custom low-precision layer (added to tensorflow by compiling C++ code and attaching the generated .s0 file to the python codes).

This effectively models the low precision convolution that we do DLA. Exact implementation is `lp_op()` method shown below.

To model saturation after bias add, `SW RefModel` has another custom layer called `saturate`. It saturate values in the range of (-128,127). One level of saturation is also done in `lp_op.lp()`. More on this is explained in section 4.

```
@layer
def conv(self, input, k_h, k_w, c_o, s_h, s_w, name,
   bw=cfg.WORD_WIDTH, fl=10, rs=0, biased=True,relu=
 True, padding=DEFAULT_PADDING):
  ...
  if self.isHardware:
    convolve = lambda i, k: self.lp_conv(i, k, s_h,
 s_w, bw, fl, rs, padding)   # DLA
  else:
    convolve = lambda i, k: tf.nn.conv2d(i, k, [1,
 s_h, s_w, 1], padding=padding) # Tensorflow

  with tf.variable_scope(name) as scope:
    ...
    if biased:
      conv = convolve(input, kernel)
      if relu:
        bias = tf.nn.bias_add(conv, biases)
        if self.isHardware:
          bias_s = self.saturate(bias, cfg.
 WORD_WIDTH)  # New addition for saturation
          return tf.nn.relu(bias_s)
        return tf.nn.relu(bias)
      bias_add = tf.nn.bias_add(conv, biases)
      if self.isHardware:
        return self.saturate(bias_add, cfg.
 WORD_WIDTH)  # New addition for saturation
      return bias_add
    else:
      conv = convolve(input, kernel)
      if relu:
        return tf.nn.relu(conv)
      return conv

def lp_conv(self, input, k, s_h, s_w, bw, fl, rs,
 padding):
  """
  Low precision convolution.
  """
  c = tf.nn.conv2d(input, k, [1, s_h, s_w, 1],
 padding=padding)
  return lp_op.lp(c, bw, rs)  # Do the right shift
 here and bit truncation and satturation here !
```

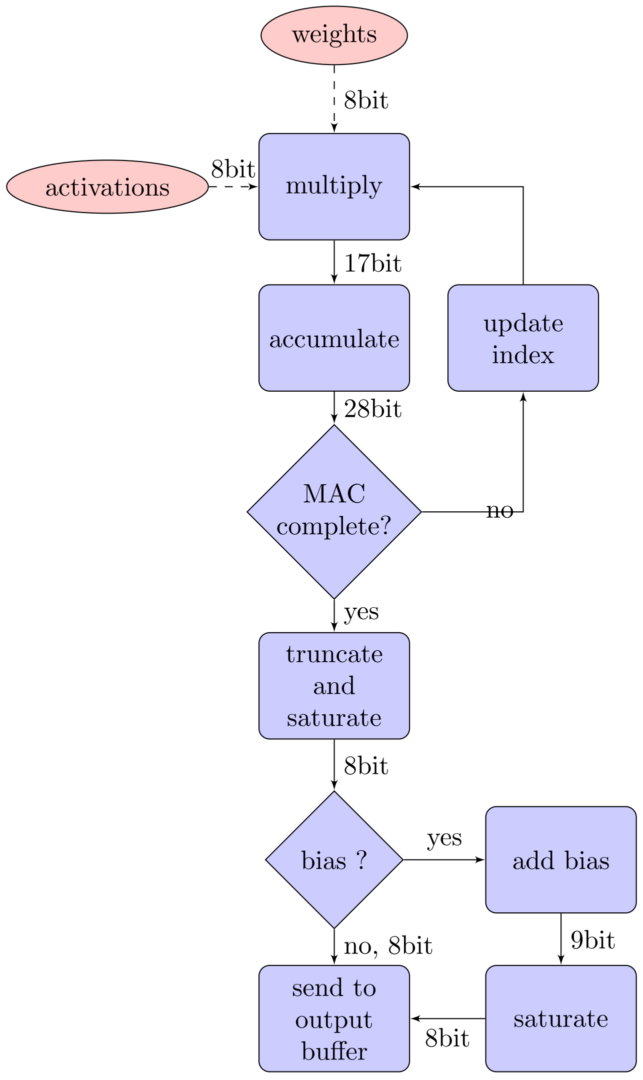
3.1.3 Fully connected

3.1.4 Get layer output

4 Custom Layers for DLA modules

`SW RefModel` mimics DLA hardware by using some custom layers along with standard Tensorflow layers. In this section we will go through the process of adding new layers with a case study of saturation. DLA operates on 8bit input and output pixels/weights.

The flow diagram of DLA MAC is shown below. As can be observed, the accumulation output during convolution can be as large as 28bits. But since MAC outputs are 8bit wide, so we do truncation and saturation. Similarly, after adding bias to the MAC output, we get a 9bit output. We do saturation here to get the final 8bit output which is pushed to output buffer.



4.1 Saturation modeling

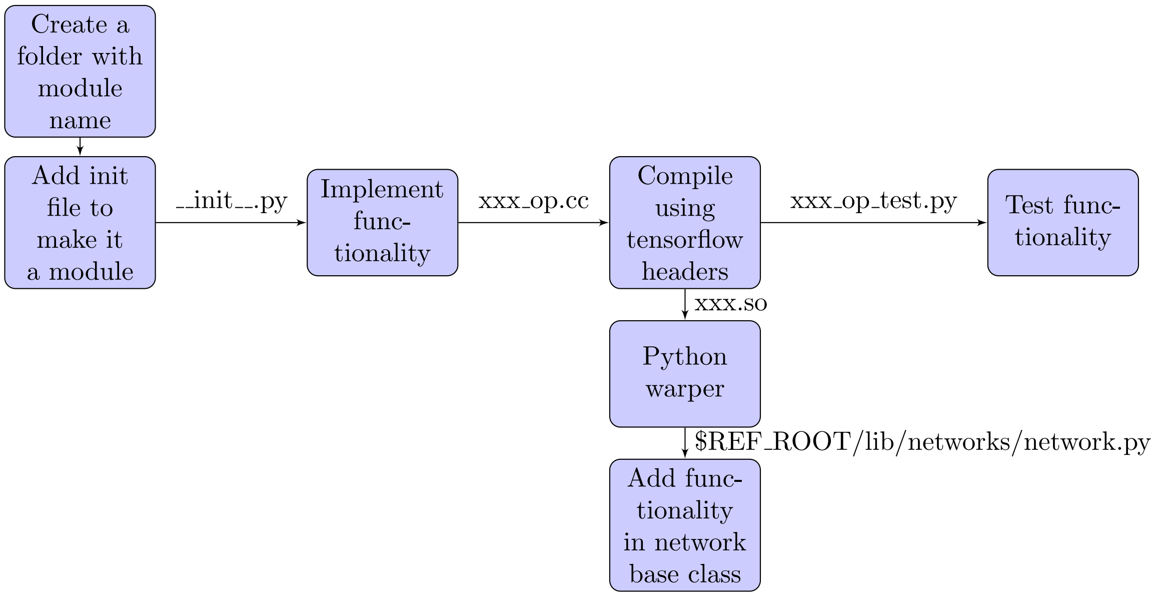
Outputs in DLA are 8bit wide (including sign bit). So all outputs are saturateed to be in (-128,127). Anything $\geq$ 127 is made 127 and anything $\leq$ 128 is made -128. This is done after adding the bias values to the truncated MAC output.

To mimic this behavior, `SW RefModel` uses custom layer implemented in C++ (with Tensorflow headers). The files related to this can be found out at `$REF_ROOT/lib/saturate`. The user should see the following files in this directory:

```
$REF_ROOT/lib/saturate
├── __init__.py
├── saturate_op.cc - C++ implementation of saturation.
├── sp.so
└── saturate_op.py - Python module to load the .so file.
```

The core logic is implemented in `saturate_op.cc` file. `sp.so` is the output when we compile `saturate_op.cc`. `saturate_op.py` file loads `sp.so` into python environment and treats it as a Tensorflow layer. `__init__.py` make saturate folder a module that can be imported in `SW RefModel`.

The steps to create a new custom tensorflow layer and make it a Python module for software reference model is shown below:


Create a folder with module name
Add init file to make it a module
__init__.py
Implement functionality
xxx_op.cc
Compile using tensorflow headers
xxx_op_test.py
Test functionality
xxx.so
Python warper
$REF_ROOT/lib/networks/network.py
Add functionality in network base class

4.1.1 saturate_op.cc

The core logic to mimic hardware behavior is implemented in this file. `SW RefModel` uses Tensorflow's ability to add custom layers to achieve this in an efficient manner. The C++ files uses the standard template[3]. The core logic for saturation is explained next with code and comments below.

```
void Compute(OpKernelContext* context) override {
  ...
  // Flatten output the output tensor for easy access
   and indexing
  auto output_flat = output_tensor->flat<T>();

  // Based on the bit width, max and min values for
   saturation are decided.
  T max_val;
  T min_val;
  if ( bw_ == 8 ){  // 8 bits saturation
    max_val = (T)127.0;
    min_val = (T)-128.0;
  }
  else {    // 16 bit saturation
    max_val = (T)32767;
    min_val = (T)-32768.0;
  }

  // Get the input size to determine when to stop
   iterating. Number of inputs == Number of outputs
  const int N = input.size();
  // Iterate through all the inputs
  for (int i = 0; i < N; i++) {
    // If value is greater than max value, saturate to
    max
    if (input(i) > max_val)
      output_flat(i) = max_val;
    // If value is less than min value, saturate to
   min
    else if (input(i) < min_val)
      output_flat(i) = min_val;
    // If value is within 8bit range, do nothing !
    else
      output_flat(i) = input(i);
  }
}
```

[3]The user should just focus of modifying the compute and REGISTER_OP function to quickly implement a new layer. Most of the template can be used as is.

4.1.2 Compiling the new Tensorflow Op

For Tensorflow to add the op to its own list of ops, the user needs to compile the C++ codes with Tensorflow's libs. For that the template shown below can be used[4].

```
#!/usr/bin/env bash
TF_INC=$(python -c 'import tensorflow as tf; print(tf.
    sysconfig.get_include())')
# For older versions of tensorflow the command shown
    should be used.
#TF_LIB=$(python -c 'import tensorflow as tf; print(tf
    .sysconfig.get_lib())')

echo $TF_INC
g++ -std=c++11 -shared -D_GLIBCXX_USE_CXX11_ABI=0 -o
    lp.so lp_op.cc \
  -I $TF_INC -fPIC #-L $TF_LIB -ltensorflow_framework
```

4.2 Truncation and saturation modeling

MAC accumulator output in DLA goes through a series of processing before being added to the `bias` value and then sent to output buffer. These steps are shown below:

```
#include <stdio.h>
#include <cfloat>

#include "third_party/eigen3/unsupported/Eigen/CXX11/
    Tensor"
#include "tensorflow/core/framework/op.h"
#include "tensorflow/core/framework/op_kernel.h"
#include "tensorflow/core/framework/tensor_shape.h"
#include "tensorflow/core/framework/shape_inference.h"

using namespace tensorflow;
typedef Eigen::ThreadPoolDevice CPUDevice;
```

[4]The user needs to change lp.sp and lp_op.so to appropriate filenames.

```
REGISTER_OP("Lp")
  .Attr("T: {float, double}")
  .Attr("bw: int")  // Bit width value, currently
   supported, 8bit and 16bit
  .Attr("rs: int")  // Right shift value
  .Input("bottom_data: T")
  .Output("top_data: T")
  .SetShapeFn([](::tensorflow::shape_inference::
   InferenceContext* c) {
    c->set_output(0, c->input(0));
    return Status::OK();
  });

template <typename Device, typename T>
class LpOp : public OpKernel {
  public:
    explicit LpOp(OpKernelConstruction* context) :
   OpKernel(context) {
      OP_REQUIRES_OK(context, context->GetAttr("bw", &
   bw_));
      OP_REQUIRES(context, bw_>= 0, errors::
   InvalidArgument("Need bw >= 0, got ",bw_));
      OP_REQUIRES_OK(context, context->GetAttr("rs", &
   rs_));
    }

  void Compute(OpKernelContext* context) override {
    // Grab the input tensor
    const Tensor& input_tensor = context->input(0);
    auto input = input_tensor.flat<T>();

    // Create an output tensor
    Tensor* output_tensor = NULL;
    OP_REQUIRES_OK(context, context->allocate_output
   (0, input_tensor.shape(),
                                                      &
   output_tensor));
    auto output_flat = output_tensor->flat<T>();

  T max_val;
  T min_val;

  if ( bw_ == 8 ){  // 8 bits saturation
    max_val = (T)127.0;
    min_val = (T)-128.0;
  }
  else {     // 16 bit saturation
    max_val = (T)32767;
    min_val = (T)-32768.0;
  }

    const int N = input.size();
    for (int i = 0; i < N; i++) {

    int shifted_out = int(input(i)) >> rs_; // Shift
   out rs bits
    int sliced_out = shifted_out % int(pow(2.0,bw_-1))
   ; // slice bitwidth -1 bits from lsb (1 bit for
   sign)

    if (shifted_out < int(min_val)) // Saturate to max
     value
      output_flat(i) = min_val;
    else if (shifted_out > int(max_val))  // Saturate
   to min value
      output_flat(i) = max_val;
    else
      output_flat(i) = (T)sliced_out;   // bit slice
   and send out
    }
  }
  private:
    int bw_;
    int rs_;
};

//REGISTER_KERNEL_BUILDER(Name("Lp").Device(DEVICE_CPU
   ), LpOp);
REGISTER_KERNEL_BUILDER(Name("Lp").Device(DEVICE_CPU).
   TypeConstraint<float>("T"), LpOp<CPUDevice, float>)
   ;
REGISTER_KERNEL_BUILDER(Name("Lp").Device(DEVICE_CPU).
   TypeConstraint<double>("T"), LpOp<CPUDevice, double
   >);
```

Importing c++ module into tensorflow and python

```
import tensorflow as tf
import os.path as osp
```

```
filename = osp.join(osp.dirname(__file__), 'lp.so')
_lp_module = tf.load_op_library(filename)
lp = _lp_module.lp
```

testing the module

```
import tensorflow as tf
import numpy as np
import lp_op

def getFixedPoint(num, totalBits, fractionBits, mode
    =0):
  """
  Returns a fixed point value of num.
  num - input number
  totalBits - total number of bits
  fractionBits - number of fractional bits
  mode - 0: returns str, 1: returns float
  """
  if isinstance(num, basestring):
    num = float(num)
  sign = 1
  if num < 0:
    sign = -1
  if mode == 1:
    return sign*round(abs(num)*pow(2,fractionBits))/
   pow(2,fractionBits)
  return str(sign*round(abs(num)*pow(2,fractionBits))/
   pow(2,fractionBits))

# TESTBENCH TO TEST THE OP
array =  -9800

data = tf.convert_to_tensor(array, dtype=tf.float32)
yy = lp_op.lp(data, 7, 3)
init = tf.global_variables_initializer()

## Launch the graph.
sess = tf.Session(config=tf.ConfigProto(
    log_device_placement=True))
y1 = sess.run(yy)

print 'output : ' + str(y1)
```

compiling

```
#!/usr/bin/env bash
TF_INC=$(python -c 'import tensorflow as tf; print(tf.
   sysconfig.get_include())')
# For older versions of tensorflow the command shown
   should be used.
#TF_LIB=$(python -c 'import tensorflow as tf; print(tf
   .sysconfig.get_lib())')

echo $TF_INC
g++ -std=c++11 -shared -D_GLIBCXX_USE_CXX11_ABI=0 -o
   lp.so lp_op.cc \
  -I $TF_INC -fPIC #-L $TF_LIB -ltensorflow_framework
```