BMNet Library User Guide

BITMAIN

SOPHON

2018-10-09

Contents

1 Introduction

The BMNET library is designed to convert the neural networks defined by CAFFE to target instructions. It seems like a compiler which translates high-level language into machine instructions. It also contains three phase which are the front end, the optimizer and the back end. The front end parses source code, extracts network prototxt and weights. The optimizer is responsible for doing a broad variety of transformations to try to improve the code's running time. The back end (also known as the code generator) then maps the code onto the target instruction set. In the BM1880 platform,we add a new feature called INT8 computation,it can provide better performance such as inference speedup. INT8 computation need an calibration table to modify network parameter,you can refer section 2 for how to generate a network's calibration table. Refer this document,you can convert a network from FP32 to INT8 without significant accuracy loss.

2 General Description

We provide multiple utility tools to convert CAFFE models into machine instructions. These instructions, as well as model's weights, would be packed into a file named bmodel (model file for BITMAIN targets), which can be executed in BITMAIN board directly. BMNet has implemented many common layers, the full list of build-in layers is in below table, and many more layers are in developing:

- Activation
- BatchNorm
- Concat
- Convolution
- Eltwise
- Flatten
- InnerProduct
- Join
- LRN
- Normalize
- Permute
- Pooling
- PReLU
- PriorBox
- Reorg
- Reshape
- Scale
- Split
- Upsample

If layers of your network model are all supported in BMNet, it is very convenient to use command line to compile the network, otherwise you can refer to Chapter 3 to add customized layers.

2.1 Programing model

The BMNET library offers a set of API and tool to convert caffemodel into machine instructions, which is saved in bmodel file. The bmodel file also keeps more information of network model, such as network name, target name, shape, weight, etc.

2.2 Quick Start

This section show you how to transform FP32 caffemodel to INT8 quickly. Prepare the following:

1. a trained CAFFE model:deploy.prototxt and alexnet.caffemodel (files in /mybmnet/-data/alexnet/)
2. calibration dataset:ilsvrc12_val_lmdb and mean.binaryproto (files in /mybmnet/-data/alexnet/)
3. calibration tool:calibration_caffe.bin
4. BMNET tool:bm_builder.bin

Remark:in this document,caffemodel in the path /mybmnet/data/xxxnet, calibration out in the path /mybmnet/out/xxxnet, bmkernel and bmruntime library in /mybmnet/lib.

2.2.1 Operation steps

Step 1. Use calibration tool to generate INT8 caffemodel.

```
$ calibration_caffe.bin alexnet /mybmnet/data/alexnet/ /mybmnet/out
   /alexnet/
```

You will get the calibration output file in /mybmnet/out/alexnet/.

Step 2. Use BMNET tool to generate bmodel file.

```
$ bm_builder.bin -t bm1880 -n alexnet -c bmnet_alexnet_int8.1x10.
   caffemodel --in_ctable=bmnet_alexnet_calibration_table.1x10.pb2
   --out_ctable=alexnet_ctable_opt.pb2 --enable-weight-optimize=yes
    -u /mybmnet/lib -s 1,3,227,227 -p alexnet_frontend_opt.proto -o
    alexnet_1_bmodel_int8.bin
```

2.2.2 Output

Follow the operation steps,you will get the cmdbuf file “alexnet_cmdbuf_int8.bin” ,run on BITMAIN board,enjoy it.

2.3 Requirement

You need to install bmkernel & bmruntime library firstly.

3 BMENT C++ API References

3.1 TensorOp

TensorOp represents a BMNET IR, which is a bridge between front end and back end. it provides lots of member method to set information to or get from it. Below is the prototype: namespace bmnet {

```
class TensorOp {
public:
  int input_shape_size();
  int output_shape_size();
  const TensorShape& input_shape(int index);
  const TensorShape& output_shape(int index);
  TensorShape* add_output_shape();
  u64 global_input(int index);
  u64 global_output(int index);
  TGCustomizedParameter* mutable_tg_customized_param();
  const TGCustomizedParameter& tg_customized_param();
};
}
```

3.1.1 TensorOp::input_shape_size

```
void TensorOp::input_shape_size()
```

Return the number of inputs.

3.1.2 TensorOp::output_shape_size

```
void TensorOp::output_shape_size()
```

Return the number of outputs.

3.1.3 TensorOp::input_shape

```
const TensorShape& TensorOp::input_shape(
    int index)
```

Return shape of input by index.

Parameter	Type	Description
index	int	[Required] index of input that to be returned.

3.1.4 TensorOp::output_shape

```
const TensorShape& TensorOp::output_shape(
    int index)
```

Return shape of output by index.

Parameter	Type	Description
index	int	[Required] index of output that to be returned.

3.1.5 TensorOp::add_output_shape

```
TensorShape* TensorOp::add_output_shape()
```

Return a mutable pointer to a new added TensorShape of outputs. The returned TensorShape could be modified latter.

3.1.6 TensorOp::global_input

```
u64 TensorOp::global_input(
    int index)
```

Return offset of input tensor by index, while it was stored in device memory.

Parameter	Type	Description
index	int	[Required] index of input that to be returned.

3.1.7 TensorOp::global_output

```
u64 TensorOp::global_output(
    int index)
```

Return offset of output tensor by index, while it was stored in device memory.

Parameter	Type	Description
index	int	[Required] index of output that to be returned.

3.1.8 TensorOp::mutable_tg_customized_param

```
TGCustomizedParameter* TensorOp::mutable_tg_customized_param()
```

Return a mutable pointer to parameters of customized BMNET IR.

3.1.9 TensorOp::tg_customized_param

```
const TGCustomizedParameter& TensorOp::tg_customized_param()
```

Return reference of customized BMNET IR's paramters.

3.2 CustomizedCaffeLayer

CustomizedCaffeLayer is abstract class, which is used to implement a Layer to convert CAFFE Layer into BMNet IR(please refer to Chapter 5 for details about BMNet IR). If you want to introduce a customized CAFFE layer into BMNet, please inherit this class and implement all pure virtual functions of it. The CustomizedCaffeLayer inherits from CaffeLayer/Layer class. Below are the prototypes of them:

```
namespace bmnet {

```

```
3  class Layer {
4  public:
5    Layer();
6    virtual ~Layer(void);
7    virtual std::string layer_name() = 0;
8    virtual void dump() = 0;
9    virtual void codegen(TensorOp *op) = 0;
10
11 protected:
12   void add_output_offset(int offset);
13 };
14 }
15
16 namespace bmnet {
17
18 class CaffeLayer : public Layer {
19 public:
20   CaffeLayer(){}
21   virtual ~CaffeLayer(void);
22 protected:
23   caffe::LayerParameter &layer_;
24 };
25 }
26
27 namespace bmnet {
28
29 class CustomizedCaffeLayer : public CaffeLayer {
30 public:
31   CustomizedCaffeLayer();
32   ~CustomizedCaffeLayer();
33   void setup(TensorOp* op) override {
34 ...
35 ...
36     TGCustomizedParameter* param = op->mutable_tg_customized_param
          ();
37     param->set_sub_type(layer_name());
38   }
39 };
40 }
```

3.2.1 CustomizedCaffeLayer::layer_name

```
std::string CustomizedCaffelayer::layer_name()
```

Pure virtual function, return type of new added CAFFE layer.

3.2.2 CustomizedCaffeLayer::dump

```
void CustomizedCaffelayer::dump()
```

Pure virtual function, is used to print information of CAFFE Layer.

3.2.3 CustomizedCaffeLayer:: setup

```
void CustomizedCaffelayer::setup()
```

Option. It is used to set sub type of Customized Layer only. Implement by default. If child class will override it, this parent class setup function must be call first.

3.2.4 CustomizedCaffeLayer::codegen

```
void CustomizedCaffelayer::codegen (
TensorOp* op)
```

Pure virtual function, is used to setup BMNET IR according to LayerParameter of CAFFE Layer. In this function, you should setup output shape and fill parameters to TensorOp.

Parameter	Type	Description
op	TensorOp*	[Required] pointer to a instance of BMNET IR

3.2.5 CustomizedCaffeLayer::add_output_offset

```
void CustomizedCaffelayer::add_output_offset (
int offset)
```

Protected member method, should be called when setup output offset of Layer's top.

Parameter	Type	Description
offset	int	[Required] offset of output, should be 0.

3.2.6 CustomizedCaffeLayer::layer_

```
caffe::LayerParameter CustomizedCaffelayer::&layer_
```

Protected member variable, which is reference of customized CAFFE layer's LayerParameter.

3.3 CustomizedTensorFixedInst

CustomizedTensorFixedInst is abstract class, which is used to implement a Layer to convert BMNET IR into instructions by BMKernel APIs. Please inherit this class and implement all pure virtual functions of it. The CustomizedTensorFixedInst inherits from TensorFixedInst/ TensorInst class. Below are the prototypes of them:

```
namespace bmnet {
class TensorFixedInst: public TensorInst {
public:
  TensorFixedInst() : TensorInst() {}
  TensorFixedInst(TensorOp &op) : TensorInst(op) {}
  virtual ~ TensorFixedInst (void);
  void SetCalibrationParameter(
      const LayerCalibrationParameter &calibration_parameter) {
    m_calibrationParameter = calibration_parameter;
  }
  void AddInputCalibrationParameter(
      const LayerCalibrationParameter &calibration_parameter) {
    m_inputCalibrationParameter.push_back(calibration_parameter);
  }
protected:
  LayerCalibrationParameter m_calibrationParameter;
  std::vector<LayerCalibrationParameter>
     m_inputCalibrationParameter;
};
}
```

```
namespace bmnet {

```

```
3  class TensorInst {
4  public:
5    TensorInst();
6    virtual ~TensorInst(void);
7    virtual std::string inst_name() = 0;
8    virtual void dump() = 0;
9    virtual void encode() = 0;
10
11 protected:
12   TensorOp &op_;
13 };
14 }
```

```
1  namespace bmnet {
2
3  class CustomizedTensorFixedInst : public TensorFixedInst {
4  public:
5    CustomizedTensorFixedInst ();
6    ~CustomizedTensorFixedInst ();
7  protected:
8    u64 get_global_neuron_base();
9    u64 get_global_weight_base();
10 };
11 }
```

3.3.1 CustomizedTensorFixedInst::inst_name

```
1  std::string CustomizedTensorFixedInst::inst_name()
```

Pure virtual function, return type of customized BMNET IR.

3.3.2 CustomizedTensorFixedInst::dump

```
1  void CustomizedTensorFixedInst::dump()
```

Pure virtual function, is used to print information of BMNET IR.

3.3.3 CustomizedTensorFixedInst::encode

```
void CustomizedTensorFixedInst::encode()
```

Pure virtual function, is used to convert BMNET IR into instructions using BMKernel APIs.

3.3.4 CustomizedTensorFixedInst::get_global_neuron_base

```
u64 CustomizedTensorFixedInst::get_global_neuron_base()
```

Protected member method, return the base address, where the neurons are stored in device memory.

3.3.5 CustomizedTensorFixedInst::get_global_weight_base

```
u64 CustomizedTensorFixedInst::get_global_weight_base()
```

Protected member method, return the base address, where weight is stored in device memory.

3.3.6 CustomizedTensorFixedInst::op_

```
TensorOp CustomizedTensorFixedInst::&op_
```

Protected member variable, which is reference of BMNET IR.

3.4 TGCustomizedParamter

TGCustomizedParamter represents a customized BMNET IR's parameters. It provides member methods to set parameters to or get from it. Below is the prototype:

```
namespace bmnet {

class TGCustomizedParameter {
public:
  int i32_param_size();
  int f32_param_size();
  int i32_param(int index);
  float f32_param(int index);
  void add_i32_param(int value);
  void add_f32_param(float value);
```

```
11 };
12 }
```

3.4.1 TGCustomizedParamter::i32_param_size

```
1 void TGCustomizedParamter::i32_param_size()
```

Return the number of int parameters, which stored in TGCustomizedParamter.

3.4.2 TGCustomizedParamter::f32_param_size

```
1 void TGCustomizedParamter::f32_param_size()
```

Return the number of float parameters, which stored in TGCustomizedParamter.

3.4.3 TGCustomizeParamter::i32_param

```
1 int TGCustomizedParamter::i32_param(
2     int index)
```

Return int parameter by index.

Parameter	Type	Description
index	index	[Required] index of int parameter that to be returned.

3.4.4 TGCustomizeParamter::f32_param

```
1 float TGCustomizedParamter::f32_param(
2     int index)
```

Return int parameter by index.

Parameter	Type	Description
index	index	[Required] index of float parameter that to be returned.

3.4.5 TGCustomizeParamter::add_i32_param

```
void TGCustomizedParamter::add_i32_param(
    int value)
```

Append a new int parameter to TGCustomizedParamter.

Parameter	Type	Description
value	int	[Required] int parameter.

3.4.6 TGCustomizeParamter::add_f32_param

```
void TGCustomizedParamter::add_f32_param(
    int value)
```

Append a new int parameter to TGCustomizedParamter.

Parameter	Type	Description
value	float	[Required] float parameter.

3.5 TensorShape

TensorShape represents a shape of tensor. Below is the prototype:

```
namespace bmnet {

class TensorShape {
public:
  void CopyFrom(const TensorShape& from);
  int dim_size() const;
  int dim(int index);
  void add_dim(int value);
};
}
```

3.5.1 TensorShape::dim_size

```
void TensorShape::dim_size()
```

Return the number of dims.

3.5.2 TensorShape::dim

```
int TensorShape::dim(
    int index)
```

Return one dim by index.

Parameter	Type	Description
index	int	[Required] index of dim that to be returned.

3.5.3 TensorShape::add_dim

```
void TensorShape::add_dim(
    int value)
```

Append a dim to TensorShape.

Parameter	Type	Description
value	int	[Required] new dim to be appended.

3.5.4 TensorShape::CopyFrom

```
void TensorShape::CopyFrom(
    const TensorShape& from)
```

Copy from another TensorShape instance.

Parameter	Type	Description
value	const TensorShape&	[Required] source TensorShape instance.

3.6 CaffeBuilder

CaffeBuilder is a class, which provides a uniform interface to combine front end/optimizer/back end core code into one, to compile CAFFE neuron network graph into bmodel file. The CaffeBuilder inherits from Builder class, which is a base compiler class. Below are the prototypes of them:

```
namespace bmnet {

class Builder {
public:
  Builder(CHIP_VER ver);
  virtual ~Builder();
  void addCustomizedTensorInst(TensorInst *inst);
  void build(int n, int c, int h, int w, int opt);
  void store_prototxt(const char *dst);
  void store_model(const char *net_name, const char *dst);
};
}
```

```
namespace bmnet {

class CaffeBuilder : public Builder {
public:
  CaffeBuilder(CHIP_VER ver, const char *modified_proto,
        const char *caffemodel, const char *weight_bin,
        const char *in_ctable, const char *out_ctable);
  ~CaffeBuilder();
  void addCustomizedLayer(Layer *layer);
};
```

3.6.1 CaffeBuilder::CaffeBuilder

```
CaffeBuilder::CaffeBuilder(
CHIP_VER ver,
const char *modified_proto,
const char *caffemodel,
const char *weight_bin,
const char *in_ctable,
const char *out_ctable)
```

Constructor function of CaffeBuilder class.

Parameter	Type	Description
ver	CHIP_VER	[Required] The target chip version. Currently only BM_CHIP_BM1880 is available.
modified_proto	const char*	[Optional] The modified prototxt file, please refer Chapter 4 to get more detail.
caffemodel	const char*	[Required] The specified caffemode file of network
weight_bin	const char*	[Optional] The specified weight file of network
in_ctable	const char*	[Required] The specified calibration table file of network
out_ctable	const char*	[Required] The specified weight file of network

modified_proto are optional parameters, that means you no need to fill all of this parameters. Below combination are valid: 1) caffemodel only; 2) caffemodel, as well as modified_protos

3.6.2 CaffeBuilder::Builder

```
1 void CaffeBuilder::build(
2 int n,int c,int h ,int w,int opt)
```

Core member function of CaffeBuilder class, used to compile the network by specifying input shape and optimization level.

Parameter	Type	Description
n, c, h, w	int	[Required] The input shape

Parameter	Type	Description
opt	int	[Optional] The input optimization options. The default value is BM_OPT_LAYER_GROUP_WITH_WEIG

Below are the values for opt.

Value	Description
OPT_NONE	No optimization
BM_OPT_LAYER_GROUP	Divides layers into clusters to optimize the bandwidth overhead.
BM_OPT_LAYER_GROUP_WITH_WEIG	Add additional optimization to reduce the device memory footprint and reshape weight.

3.6.3 CaffeBuilder::store_prototxt

```
1 void CaffeBuilder::store_prototxt(
2 const char* dst)
```

store the optimized network graph as a file.

Parameter	Type	Description
dst	const char*	[Required] File to be stored.

3.6.4 CaffeBuilder::store_model

```
1 void CaffeBuilder::store_model(
2 const char* net_name,
3 const char* dst,
4 onst char* plugin_path=nullptr)
```

Store compiled instructions, weight and other information of the network as a bmodel file.

Parameter	Type	Description
net_name	const char*	[Required] The network name.
dst	const char*	[Required] File to store bmodel.
plugin_path	const char*	[optional] cpu op plugins.

3.6.5 CaffeBuilder::addCustomizedLayer

```
1 void CaffeBuilder::addCustomizedLayer(
2 Layer* layer)
```

Register a new added customized layer, which used to convert CAFFE layer into BMNet IR (Intermediate representation).

Parameter	Type	Description
Layer	Layer*	[Required] pointer to instance of Class Layer.

3.6.6 CaffeBuilder::addCustomizedTensorInst

```
1 void CaffeBuilder::addCustomizedTensorInst(
2 TensorInst* inst)
```

Register a new added customized TensorInst (Tensor Instruction), which used to convert BMNet IR into instructions.

Parameter	Type	Description
inst	TensorInst*	[Required] pointer to instance of Class Layer.

4 Tools

4.1 calibration_caffe

4.1.1 Description

The tool “calibration_caffe” takes a caffemodel file and training data to convert a new INT8 caffemodel file and an calibration table file.These two files are the input of BMNET,and make INT8 computation on BM1880 is possible.

4.1.2 End-user Options

```
calibration_caffe.bin [net_name] [caffemodel_path] [INT8_model_path
    ]
```

Parameter	Description
[net_name]	Specify the name of network model,currently we support lenet, vgg16, resnet(xx), mobilenet(xx), densenet, alexnet, googlenet(v1/v3), mtcnn, SSD_300x300, tiny_ssh.
[caffemodel_path]	Specify the input path of network model,a trained caffemodel and training data should be included in this path.
[INT8_model_path]	Specify the output path of network model,an INT8 caffemodel and an calibration table will be generated.

example:

```
$ calibration_caffe.bin alexnet /mybmnet/data/alexnet/ /mybmnet/out
    /alexnet/
```

4.2 bm_builder.bin

4.2.1 Description

The bm_builder.bin combines frontend, optimizer and backend modules into one executable binary, and links to libbmnet.so. It takes network's caffemodel and deploy.prototxt as inputs, and finally generates bmodel after compiled.

4.2.2 End-user Options

```
bm_builder.bin
  -t or --target
    Specify the name of BITMAIN target board (bm1880).
  -n or --name
    Specify the name of deep learning network.
  -s or --shape
    Specify the input shape of network. Dims should be separated by
        commas and no backspace is allowed.
  -u or --plugin
    Specify the directories of cpu op plugins.
  -c or --caffemodel
    Specify the caffemodel generated by calibration_caffe.bin.
  -m or --modified_proto
    If you want to modify the prototxt, specify the modified deploy
       .prototxt of network.
  -o or --out_model
    Specify the output bmodel file.
  -d or --in_ctable=input_ctable_file
    Specify the calibration table generated by calibration_caffe.
       bin.
  -e or --out_ctable=output_ctable_file
    Specify the output of optimizer calibration table.
  -p or --out_proto=output_prototxt_file
    Specify the optimizer prototxt file of network.
  --enable-weight-optimize=yes|no [no]
    Specify the option(yes or no) to enable or disable optimization
       .
```

4.2.3 Key API

4.2.3.1 CaffeBuilder

Constructor.You can refer section 3 for more information.

4.2.3.2 builder

Generate bmodel.You can refer section 3 for more information.

4.2.3.3 store_model

Store bmodel file.You can refer section 3 for more information.

4.3 Example

Here is an example to convert vgg16 caffe model to a bmodel. You need to prepare the dataset for calibration first at path/for/dataset.

```
1 $ calibration_caffe.bin vgg16 path/for/dataset --out path/for/
    calibration/output
2 $ bm_builder.bin -t bm1880 -n vgg16 -c bmnet_vgg16_int8.1x10.
    caffemodel --in_ctable=bmnet_vgg16_calibration_table.1x10.pb2 --
    out_ctable=vgg16_bmodel_ctable_opt.pb2 --enable-weight-optimize=
    yes -s 1,3,224,224 -p vgg16_bmodel_opt.proto -o
    vgg16_1_3_224_224.bmodel
```

bmnet_vgg16_int8.1x10.caffemodel and bmnet_vgg16_calibration_table.1x10.pb2 is generated by calibration_caffe.bin.

5 IR Parameter

The IR parameter is almost the same as the CAFFE layer parameter, if you are familiar with the CAFFE layer parameter, you will not get lost with bmnet IR parameter. We do some modification to the CAFFE proto to make it convenient for our implementation. You can review the below section for more information.

5.1 Layer Parameter

Layer Parameter is the base class of all kinds of IRs in BMNET, like convolution or pooling IR. We put some common parameters like the IR name, IR type, input shape, output shape etc in the base class and keep the IR's individual feature in the sub layer parameter. Below is the description of the message format of the layer parameter.

```
message LayerParameter {
  optional string name = 1;
  optional string type = 2;
  repeated string bottom = 3;
  repeated string top = 4;
  optional Phase phase = 10;
  repeated BlobProto blobs = 7;
  repeated BlobShape input_shape = 200;
  repeated BlobShape output_shape = 201;
  repeated uint64 global_input = 202;
  repeated uint64 global_output = 203;
  optional StartParameter start_param = 210;
  optional EndParameter end_param = 211;
  optional TGConvolutionParameter tg_convolution_param = 260;
  optional TGPoolingParameter tg_pooling_param = 261;
  optional TGInnerProductParameter tg_inner_product_param = 262;
  optional TGActivationParameter tg_activation_param = 263;
  optional TGBatchNormParameter tg_batchnorm_param = 264;
  optional TGSoftmaxParameter tg_softmax_param = 265;
  optional TGDropoutParameter tg_dropout_param = 266;
  optional TGEltwiseParameter tg_eltwise_param = 267;
```

```
22    optional TGLRNParameter tg_lrn_param = 268;
23    optional TGPowerParameter tg_power_param = 269;
24    optional TGScaleParameter tg_scale_param = 270;
25    optional TGPReLUParameter tg_prelu_param = 271;
26    optional TGUpsampleParameter tg_upsample_param = 272;
27    optional TGReorgParameter tg_reorg_param = 273;
28    optional TGConcatParameter tg_concat_param = 274;
29    optional TGPermuteParameter tg_permute_param = 275;
30    optional TGNormalizeParameter tg_norm_param = 276;
31    optional TGPriorBoxParameter tg_prior_box_param = 277;
32    optional TGCropParameter tg_crop_param = 278;
33    optional TGReductionParameter tg_reduction_param = 279;
34    optional TGShuffleChannelParameter tg_shuffle_channel_param =
          280;
35 }
```

Only some simple parameter list below:

Parameter	Type	Description
name	string(optional)	The layer name
type	string(optional)	The layer type
bottom	string(repeated)	The bottom layer name
top	string(repeated)	The top layer name
blobs	BlobProto(repeated)	The blobs containing the numeric parameters of the layer. see more detail of BlobProto in below section
input_shape	BlobShape(repeated)	The shape of the inputs, may have more than 1 input. See more detail of BlobShape in below section
output_shape	BlobShape(repeated)	The shape of the outputs, may have more than 1 output. See more detail of BlobShape in below section
lobal_input	uint64(optional)	The global memory address of the input of current layer

Parameter	Type	Description
global_output	uint64(optional)	The global memory address of the output of current layer

In below section, we will separately introduce other complex parameters appeared in layer parameter.

5.1.1 BlobShape

BlobShape is used to specify the blob shape information. See below for more information.

```
message BlobShape {
  repeated int64 dim = 1 [packed = true];
  optional bool aligned = 2 [default = false];
  optional int32 data_type_size = 3 [default = 4];
}
```

Parameter	Type	Description
dim	Int64(int64)	The dims of the shape, specify the size of each dimension
aligned	bool(optional)	Whether or not the shape need to aligned
data_type_size	int32(optional)	The data type size, for instance, if data type is float, then data_type_size is 4

5.1.2 BlobProto

BlobProto is used to specify the blob information include the blob shape and blob data. See below for more information.

```
message BlobProto {
  optional BlobShape shape = 7;
  repeated float data = 5 [packed = true];
```

```
4    repeated float diff = 6 [packed = true];
5    repeated double double_data = 8 [packed = true];
6    repeated double double_diff = 9 [packed = true];
7    optional int32 num = 1 [default = 0];
8    optional int32 channels = 2 [default = 0];
9    optional int32 height = 3 [default = 0];
10   optional int32 width = 4 [default = 0];
11   optional uint64 offset = 10 [default = 0];
12 }
```

Parameter	Type	Description
shape	BlobShape (optional)	The shape of the blob
data	float(repeated)	The place to store the data in the blob
diff	float(repeated)	The place to store the gradient data in the blob
double_data	double(repeated)	The place to store the data in the blob, in double format
double_diff	double(repeated)	The place to store the gradient data in the blob, in double format
num	int32(optional)	First dimension of the 4D, deprecated
channel	int32(optional)	Second dimension of the 4D, deprecated
eight	int32(optional)	Third dimension of the 4D, deprecated
width	int32(optional)	Fourth dimension of the 4D, deprecated
offset	int32(optional)	offset in binary weight file

5.1.3 StartParameter

StartParameter is used to specify the start parameter of the start IR in the network. See below for more information.

```
message StartParameter {
  optional uint64 input_offset = 1 [default = 0];
}
```

Parameter	Type	Description
input_offset	uint64 (optional)	The offset of the input data in global memory, not used.

5.1.4 EndParameter

EndParameter is used to specify the end parameter of the end IR in the network. See below for more information.

```
message EndParameter {
  optional uint64 output_offset = 1 [default = 0xffffffffffff];
  optional uint64 output_size = 2 [default = 0];
  optional uint64 total_neuron_size = 3 [default = 0];
}
```

Parameter	Type	Description
output_offset	uint64 (optional)	The offset of the output data in global memory
output_size	uint64 (optional)	output data size in byte
Total_neuron_size	uint64 (optional)	Total neuron size that we needed in global memory when run the network

5.1.5 TGActivationParameter

TGActivationParameter is used to specify the activation parameter of the activation IR in the network. See below for more information.

```
message TGActivationParameter {
  optional ActivationMethod activation = 5 [default = RELU];
  repeated float activation_arg = 6;
```

```
4    optional FillerParameter filler = 7;
5    optional bool channel_shared = 8 [default = false];
6    optional uint64 global_slope = 9 [default = 0xffffffffffff];
7  }
```

Parameter	Type	Description
activation	ActivationMethod (optional)	The activation method
activation_arg	float (repeated)	The activation value need in the activation layer, like the relu slope value.
filler	FillerParameter(optional)	Initial value of the learned parameter, default is 0.25 for all element.
activation_channel_shared	bool(optional)	Whether or not slope parameters are shared across channels.
global_slope	uint64(optional)	The global memory address of activation slope

5.1.6 TGBatchNormParameter

TGBatchNormParameter is used to specify the batchnorm parameter of the batchnorm IR in the network. See below for more information.

```
1  message TGBatchNormParameter {
2    optional uint64 global_mean = 5 [default = 0xffffffffffff];
3    optional uint64 global_variance = 6 [default = 0xffffffffffff];
4    optional uint64 global_fraction = 7 [default = 0xffffffffffff];
5    optional float eps = 8 [default = 1e-5];
6  }
```

Parameter	Type	Description
global_mean	uint64(optional)	The global memory address of the mean value of batchnorm layer

Parameter	Type	Description
global_variance	uint64(optional)	The global memory address of the variance value of batchnorm layer
global_fraction	uint64(optional)	The global memory address of the moving average fraction value of batchnorm layer
eps	float(optional)	Small value to add to the variance estimate so that we don't divide by zero

5.1.7 TGConcatParameter

TGConcatParameter is used to specify the concat parameter of the concat IR in the network. See below for more information.

```
message TGConcatParameter {
  optional int32 axis = 2 [default = 1];
  repeated int32 ignored_bottom = 3;
}
```

Parameter	Type	Description
axis	int32 (optional)	The axis along which to concatenate, 0 for concatenation along num and 1 for channels.
Ignored_bottom	int32 (repeated)	Specify the ignored bottom index when do concat

5.1.8 TGConvolutionParameter

TGConvolutionParameter is used to specify the convolution parameter of the convolution IR in the network. The convolution layer may fusion with other layers so the parameter list below may

contain parameters of other layers in the network. See below for more information.

```
message TGConvolutionParameter {
  optional uint64 global_weight = 5 [default = 0xffffffffffff];
  optional uint64 global_bias = 6 [default = 0xffffffffffff];
  optional uint64 global_bn_mean = 7 [default = 0xffffffffffff];
  optional uint64 global_bn_variance = 8 [default = 0xffffffffffff
     ];
  optional uint64 global_scale = 9 [default = 0xffffffffffff];
  optional uint64 global_scale_bias = 10 [default = 0xffffffffffff
     ];
  optional bool bias_term = 11 [default = false];
  repeated uint32 kernel_size = 12;
  repeated uint32 stride = 13;
  repeated uint32 dilation = 14;
  repeated uint32 pad = 15;
  repeated uint32 ins = 16;
  optional uint32 group = 17 [default = 1];
  optional bool result_add = 18 [default = false];
  optional bool do_bn = 19 [default = false];
  optional bool do_scale = 20 [default = false];
  optional bool do_scale_bias = 21 [default = false];
  optional bool do_activation = 22 [default = false];
  optional bool do_pooling = 23 [default = false];
  repeated uint32 pooling_kernel_size = 24;
  repeated uint32 pooling_stride = 25;
  repeated uint32 pooling_pad = 26;
  optional ActivationMethod activation = 27 [default = RELU];
  repeated float activation_arg = 28;
  optional uint64 global_bn_fraction = 29 [default = 0xffffffffffff
     ];
  optional float bn_eps = 30 [default = 1e-6];
  optional uint64 activation_global_slope = 31 [default = 0
     xffffffffffff];
  optional bool activation_channel_shared = 32 [default = false];
  optional bool use_winograd = 33 [default = false];
}
```

Parameter	Type	Description
global_weight	uint64(optional)	The global memory address of weight

Parameter	Type	Description
global_bias	uint64(optional)	The global memory address of bias
global_bn_mean	uint64(optional)	The global memory address of the mean value of batchnorm layer
global_bn_variance	uint64(optional)	The global memory address of the variance value of batchnorm layer
global_scale	uint64(optional)	The global memory address of scale value of scale layer
global_scale_bias	uint64(optional)	The global memory address of bias in scale layer
bias_term	bool(optional)	specifies whether to learn and apply a set of additive biases to the filter outputs
kernel_size	uint32(repeated)	specifies height and width of each filter
stride	uint32(repeated)	specifies the intervals at which to apply the filters to the input
dilation	uint32(repeated)	The dilation of convolution layer, defaults to 1
pad	uint32(repeated)	specifies the number of pixels to (implicitly) add to each side of the input
ins	uint32(repeated)	specifies the number of 0 to insert between each element of the input

Parameter	Type	Description
group	uint32(optional)	If g > 1, we restrict the connectivity of each filter to a subset of the input. Specifically, the input and output channels are separated into g groups, and the ith output group channels will be only connected to the ith input group channels.
sult_add	ool(optional)	hether need to add the original value in output
do_bn	bool(optional)	Whether need to do batchnorm layer within convolution layer
do_scale	bool(optional)	Whether need to do scale layer within convolution layer
do_scale_bias	bool(optional)	Whether need to add the bias in scale layer
do_activation	bool(optional)	Whether need to do activation layer within convolution layer
do_pooling	bool(optional)	Whether need to do pooling layer within convolution layer
pooling_kernel_size	uint32(repeated)	specifies height and width of each filter of pooling layer
pooling_stride	uint32(repeated)	specifies the intervals at which to apply the filters to the input of pooling layer

Parameter	Type	Description
pooling_pad	uint32(repeated)	specifies the number of pixels to (implicitly) add to each side of the input of pooling layer
activation	ActivationMethod(optional)	The activation method used for activation layer
activation_arg	float(repeated)	The activation value need in the activation layer, like the relu slope value.
global_bn_fraction	uint64(optional)	The global memory address of fraction in batchnorm layer
bn_eps	float(optional)	Small value to add to the variance estimate so that we don't divide by zero
activation_global_slope	uint64(optional)	The global memory address of activation slope for prelu layer
activation_channel_shared	bool(optional)	Whether or not slope parameters are shared across channels
use_winograd	bool(optional)	Whether or not to use the winograd for convolution

5.1.9 TGEltwiseParameter

TGEltwiseParameter is used to specify the eltwise parameter of the eltwise IR in the network. See below for more information.

```
message TGEltwiseParameter {
  enum EltwiseOp {
    PROD = 0;
    SUM = 1;
    MAX = 2;
  }
```

```
7    optional EltwiseOp operation = 1 [default = SUM]; //
8    repeated float coeff = 2;
9    optional bool stable_prod_grad = 3 [default = true];
10   optional bool do_activation = 4 [default = false];
11   optional ActivationMethod activation = 5 [default = RELU];
12   repeated float activation_arg = 6;
13 }
```

Parameter	Type	Description
operation	EltwiseOp(optional)	element-wise operation, default is sum operation between the input, also can set to product or max
coeff	float(repeated)	blob-wise coefficient for SUM operation
stable_prod_grad	bool(optional)	Whether to use an asymptotically slower (for >2 inputs) but stabler method of computing the gradient for the PROD operation
do_activation	bool (optional)	Whether need to do activation layer within eltwise layer
activation	ActivationMethod (optional)	The activation method, default is RELU
activation_arg	float (repeated)	The activation value need in the activation layer, like the relu slope value.

5.1.10 TGPoolingParameter

TGPoolingParameter is used to specify the pooling parameter of the pooling IR in the network. The pooling layer may fusion with activation layers so the parameter list below may contain parameters of other layers in the network. See below for more information.

```
message TGPoolingParameter {
  enum PoolMethod {
    MAX = 0;
    AVE = 1;
  }
  optional PoolMethod pool = 6 [default = MAX];
  repeated uint32 kernel_size = 7;
  repeated uint32 stride = 8;
  repeated uint32 pad = 9;
  repeated uint32 ins = 10;
  optional bool do_activation = 11 [default = false];
  optional ActivationMethod activation = 12 [default = RELU];
  repeated float activation_arg = 13;
}
```

Parameter	Type	Description
pool	PoolMethod (optional)	The pooling method, 0 is max pooling, 1 is average pooling
kernel_size	uint64(optional)	specifies height and width of each filter in pooling
stride	uint32(repeated)	specifies the intervals at which to apply the filters to the input when do pooling
pad	uint32(repeated)	specifies the number of pixels to (implicitly) add to each side of the input
ins	uint32(repeated)	specifies the number of 0 to insert between each element of the input
do_activation	bool (optional)	Whether need to do activation layer within pooling layer
activation	ActivationMethod (optional)	The activation method
activation_arg	float (repeated)	The activation value need in the activation layer, like the relu slope value.

Parameter	Type	Description

5.1.11 TGInnerProductParameter

TGInnerProductParameter is used to specify the innerproduct parameter of the innerproduct IR in the network. The innerproduct layer may fusion with activation layers so the parameter list below may contain parameters of other layers in the network. See below for more information.

```
message TGInnerProductParameter {
  optional uint64 global_weight = 5 [default = 0xffffffffffff];
  optional uint64 global_bias = 6 [default = 0xffffffffffff];
  optional bool bias_term = 7 [default = true];
  optional bool result_add = 9 [default = false];
  optional uint32 num_output = 10;
  optional bool do_activation = 11 [default = false];
  optional ActivationMethod activation = 12 [default = RELU];
  repeated float activation_arg = 13;
  optional bool weight_transpose = 14 [default = true];
  optional uint64 activation_global_slope = 31 [default = 0
     xffffffffffff];
  optional bool activation_channel_shared = 32 [default = false];
}
```

Parameter	Type	Description
global_weight	uint64(optional)	The global memory address of weight
global_bias	uint64(optional)	The global memory address of bias
bias_term	bool(optional)	specifies whether to learn and apply a set of additive biases to the filter outputs
result_add	bool(optional)	Whether need to add the original value in output
num_output	uint32(optional)	specifies the number of output for the layer

Parameter	Type	Description
do_activation	bool (optional)	Whether need to do activation layer within pooling layer
activation	ActivationMethod (optional)	The activation method
activation_arg	float (repeated)	The activation value need in the activation layer, like the relu slope value.
weight_transpose	bool (optional)	Whether or not the weight is transposed
activation_global_slope	uint64(optional)	The global memory address of activation slope if we do prelu layer within inner product layer
activation_channel_shared	bool(optional)	Whether or not slope parameters are shared across channels

5.1.12 TGLRNParameter

TGLRNParameter is used to specify the lrn parameter of the lrn IR in the network. See below for more information.

```
message TGLRNParameter {
  optional uint32 local_size = 5 [default = 5];
  optional float alpha = 6 [default = 1.];
  optional float beta = 7 [default = 0.75];
  enum NormRegion {
    ACROSS_CHANNELS = 0;
    WITHIN_CHANNEL = 1;
  }
  optional NormRegion norm_region = 8 [default = ACROSS_CHANNELS];
  optional float k = 9 [default = 1.];
  optional uint64 sqr_lut_weight = 10 [default = 0xfffffffffff];
  optional uint64 power_lut_weight = 11 [default = 0xfffffffffff];
}
```

Parameter	Type	Description
local_size	uint32(optional)	the number of channels to sum over (for cross channel LRN) or the side length of the square region to sum over (for within channel LRN)
alpha	float(repeated)	the scaling parameter
beta	float(optional)	the exponent
norm_region	NormRegion (optional)	whether to sum over adjacent channels (ACROSS_CHANNELS) or nearby spatial locations (WITHIN_CHANNEL)
k	float (optional)	value added to the divisor
sqr_lut_weight	uint64 (optional)	not used
power_lut_weight	uint64 (optional)	not used

5.1.13 TGNormalizeParameter

TGNormalizeParameter is used to specify the normalize parameter of the normalize IR in the network. See below for more information.

```
message TGNormalizeParameter {
  optional bool across_spatial = 5 [default = true];
  repeated float scale = 6;
  optional bool channel_shared = 7 [default = true];
  optional float eps = 8 [default = 1e-10];
  optional FillerParameter bias_filler = 9;
  optional uint64 global_scale = 10 [default = 0xffffffffffff];
  optional uint64 global_sqr_lut = 11 [default = 0xffffffffffff];
  optional uint64 global_sqrt_lut = 12 [default = 0xffffffffffff];
}
```

Parameter	Type	Description
across_spatial	bool(optional)	Whether or not do the normalization across all the element, default is true. If set to false, will do normalization in the channel.
scale	float(repeated)	Initial value of scale. Default is 1.0 for all
channel_shared	bool(optional)	Whether or not slope parameters are shared across channels
eps	float(optional)	Epsilon for not dividing by zero while normalizing variance
bias_filler	FillerParameter (optional)	The initialization for the learned scale parameter.
global_scale	uint64(optional)	The global memory address of scale parameter
global_sqr_lut	uint64(optional)	Not used
global_sqrt_lut	uint64(optional)	Not used

5.1.14 TGPriorBoxParameter

TGPriorBoxParameter is used to specify the priorbox parameter of the priorbox layer in the network. See below for more information.

```
message TGPriorBoxParameter {
  optional uint64 global_weight = 1 [default = 0xffffffffffff];
}
```

Parameter	Type	Description
global_weight	uint64 (optional)	The global memory address of weight

5.1.15 TGReorgParameter

TGReorgParameter is used to specify the reorg parameter of the reorg IR in the network. See below for more information.

```
message TGReorgParameter {
  repeated uint32 size = 1;
  optional uint32 stride = 2;
  optional bool reverse = 3 [default = false];
}
```

Parameter	Type	Description
size	uint32 (optional)	Not used
stride	uint32 (optional)	The stride to do reorg in h/w direction.
reverse	bool (optional)	Not used

5.1.16 TGScaleParameter

TGScaleParameter is used to specify the scale parameter of the scale IR in the network. See below for more information.

```
message TGScaleParameter {
  optional bool bias_term = 4 [default = false];
  optional FillerParameter bias_filler = 5;
  optional uint32 scale_dim = 6;
  optional uint32 inner_dim = 7;
  optional uint64 global_scale = 8 [default = 0xffffffffffff];
  optional uint64 global_bias = 9 [default = 0xffffffffffff];
  optional int32 axis = 10 [default = 1];
  optional int32 num_axes = 11 [default = 1];
  optional bool do_activation = 12 [default = false];
  optional ActivationMethod activation = 13 [default = RELU];
  repeated float activation_arg = 14;
  optional bool fusion_skipped = 15 [default = true];
}
```

Parameter	Type	Description
bias_term	bool(optional)	specifies whether to learn and apply a set of additive biases to the filter outputs
bias_filler	FillerParameter (optional)	The initialization for the learned scale parameter.
scale_dim	uint32(optional)	The dimension production from axis to num_axes
inner_dim	uint32(optional)	The dimension production from num_axes to input dimension size
global_scale	uint64(optional)	The global memory address of scale parameter
global_bias	uint64(optional)	The global memory address of bias parameter
axis	int32(optional)	The first axis of bottom[0] (the first input Blob) along which to apply
bottom[1] (the second input Blob) num_axes	int32(optional)	num_axes is determined by the number of axes by the second bottom
do_activation	bool (optional)	Whether need to do activation layer within scale layer
activation	ActivationMethod (optional)	The activation method
activation_arg	float (repeated)	The activation value need in the activation layer, like the relu slope value.
fusion_skipped	bool(optional)	Whether to skip the fusion optimization

5.1.17 TGUpsampleParameter

TGUpsampleParameter is used to specify the upsample parameter of the upsample layer in the network. See below for more information.

```
message TGUpsampleParameter {
  repeated uint32 size = 1;
}
```

Parameter	Type	Description
size	uint32 (optional)	Specify the output size after upsample, input dimension(n,c,h,w) change to output dimension (n, c, h*size, w*size).

6 Add Customized Layer

BMNET provides a serials API to add customized layers without modifying the BMNet core code. Customized layer could be a pure new layer or could be instead of original caffe layer in bmnet. Below tutorial will guide through the steps to create a simple custom layer (use LeakyRelu layer as an example, source code could be found in bmnet/example/customized_layer_1880/) instead of original caffe layer in BMNet.

6.1 Add new caffe layer definition in bmnet_caffe.proto

Modify the bmnet_caffe.proto in path "bmnet/examples/customized_layer_1880/proto". Firstly, you need to check whether the layer exist or not. If it exists skip this step, if it doesn't exist please append a new line at the end of LayerParameter with a new index and add definition of LayerParameter. Note: new layer must be added at the end line, for example, add a ReLUParameter.

```
message LayerParameter {
  optional string name = 1; // the layer name
  optional string type = 2; // the layer type
  repeated string bottom = 3; // the name of each bottom blob
  repeated string top = 4; // the name of each top blob
  ...
  optional AccuracyParameter accuracy_param = 102;
  optional EmbedParameter embed_param = 137;
  optional ExpParameter exp_param = 111;
optional ReLUParameter relu_param = 123;
}
```

```
// Message that stores parameters used by ReLU Layer
message ReLUParameter {
  optional float negative_slope = 1 [default = 0];
  enum Engine {
    DEFAULT = 0;
    CAFFE = 1;
```

```
    CUDNN = 2;
  }
  optional Engine engine = 2 [default = DEFAULT];
}
```

6.2 Add new CAFFE layer class

Create a child class that inherited from CustomizedCaffeLayer, and implement layer_name(), dump(), codegen() member methods :

- layer_name(): needs to return the string name of layer type.
- setup(): option. Only support to set set_sub_type if necessay. if not implement set_sub_type = layer type.
- dump(): dump the parameter's details of new added CAFFE layer in this function.
- codegen(): convert parameters of CAFFE layer to tg_customized_param, which is parameter of customized IR.

```
#include <bmnet/frontend/caffe/CaffeFrontendContext.hpp>
#include <bmnet/frontend/caffe/CustomizedCaffeLayer.hpp>

class LeakyReluLayer: public CustomizedCaffeLayer {
public:
  LeakyReluLayer () : CustomizedCaffeLayer() {}
  // return type name of new added CAFFE layer.
  std::string layer_name() {
    return std::string("ReLU");
  }
  // dump parameters of CAFFE layer object layer_.
  void dump() {
    const caffe::ReLUParameter &in_param = layer_.relu_param();
    float negative_slope = in_param. negative_slope();
    std::cout << "negative_slope:" << negative_slope;
  }
void setup(TensorOp* op) {
    CustomizedCaffeLayer::setup(op);
    TGCustomizedParameter* param = op->mutable_tg_customized_param
        ();
    param->set_sub_type("leakyrelu");
}
  // convert parameters of CAFFE layer to customized
  // IR(TensorOp *op)'s parameter(tg_customized_param)
```

```
24    void codegen(TensorOp *op) {
25      // get input shape
26      const TensorShape & input_shape = op->input_shape(0);
27      // get parameter from caffe proto
28      const caffe:: ReLUParameter&in_param = layer_.relu_param();
29      float negative_slope = in_param.negative_slope();
30      // set normal output shape
31      TensorShape *output_shape = op->add_output_shape();
32      output_shape->CopyFrom(input_shape);
33      // set out_param
34      TGCustomizedParameter* out_param = op->
            mutable_tg_customized_param();
35      out_param->add_f32_param(negative_slope);
36    }
37  };
```

6.3 Add new Tensor Instruction class

Create a child class that inherited from CustomizedTensorFixedInst, a class to convert IR to instructions, and implement inst_name(), dump(), encode() member functions:

- inst_name(): needs to return IR name, lowercase with prefix "tg_" + sub_type, sub_type is set at 6.2.
- dump(): dump tg_customized_param's details of IR op_.
- encode(): convert IR to instructions. If the IR could be deployed to NPU, please use BMKernel api to implement it, or you can just implement a pure CPU version used c++ language.

```
1  #include <bmnet/targets/plat-bm188x/BM188xBackendContext.hpp>
2  #include <bmnet/targets/plat-bm188x/CustomizedTensorFixedInst.hpp>
3  #include <bmkernel/bm_kernel.h>
4
5  namespace bmnet {
6
7  class TGLeakyReluFixedInst: public CustomizedTensorFixedInst {
8  public:
9    TGLeakyReluFixedInst() : CustomizedTensorFixedInst() {}
10   ~ TGLeakyReluFixedInst() {}
11   // return type name of IR
12   std::string inst_name() {
13     return std::string("tg_leakyrelu");
```

```
  }
  // dump tg_customized_param of IR op_.
  void dump() {
    const TGCustomizedParameter& param = op_.tg_customized_param();
    float alpha = param.f32_param(0);
    std::cout << "alpha:" << alpha << std::endl;
  }
  // extract parameters of tg_customized_param,
  // and implement instructions.
  void encode();
private:
    void forward(
        gaddr_t bottom_gaddr, gaddr_t top_gaddr,
        int input_n, int input_c,
        int input_h, int input_w);
};
}
```

6.3.1 NPU Version.

If the IR could be deployed to NPU, please use BMKernel APIs to implement the function encode(). More details about BMKernel APIs, please refer to related document.

```
void TGLeakyReluFixedInst::encode() {
  const TGCustomizedParameter& param = op_.tg_customized_param();
  const TensorShape& input_shape = op_.input_shape(0);
  float negative_slope = param.f32_param(0);
  assert(negative_slope > 0);
  gaddr_t input_data_gaddr     = op_.global_input(0) +
     get_global_neuron_base();
  gaddr_t output_data_gaddr    = op_.global_output(0) +
     get_global_neuron_base();
  forward (
      input_data_gaddr,
      output_data_gaddr,
      input_shape.dim(0),
      input_shape.dim(1),
      input_shape.dim(2),
      input_shape.dim(3));
}
```

```
void TGLeakyReluFixedInst::forward(
    gaddr_t                bottom_gaddr,
    gaddr_t                top_gaddr,
    int                    input_n,
    int                    input_c,
    int                    input_h,
    int                    input_w)
{
  gaddr_t slice_bottom_gaddr    = bottom_gaddr;
  gaddr_t slice_top_gaddr       = top_gaddr;
  int count                     = input_n * input_c * input_h *
      input_w;
  int slice_num                 = get_slice_num_element_wise(*_ctx,
      3, count + 1);

  int gt_right_shift_width = m_calibrationParameter.relu_param().
      gt_right_shift_width();
  int le_right_shift_width = m_calibrationParameter.relu_param().
      le_right_shift_width();
  int gt_scale = m_calibrationParameter.relu_param().gt_scale();
  int le_scale = m_calibrationParameter.relu_param().le_scale();

  for (int slice_idx = 0; slice_idx < slice_num; slice_idx++) {
      int count_sec = count / slice_num + (slice_idx < count %
          slice_num);
      // set shape
      shape_t input_shape = shape_t1(count_sec);
      tensor_lmem *bottom = _ctx->tl_alloc(input_shape, FMT_I8,
          CTRL_AL);
      tensor_lmem *relu = _ctx->tl_alloc(input_shape, FMT_I8,
          CTRL_AL);
      tensor_lmem *neg = _ctx->tl_alloc(input_shape, FMT_I8,
          CTRL_AL);
      // load
      _ctx->gdma_load(bottom, slice_bottom_gaddr, CTRL_NEURON);
      bmk1880_relu_param_t p13;
      p13.ofmap = relu;
      p13.ifmap = bottom;
      _ctx->tpu_relu(&p13);
      bmk1880_mul_const_param_t p;
      p.res_high = NULL;
      p.res_low = relu;
      p.a = relu;
```

```
36        p.b = gt_scale;
37        p.b_is_signed = true;
38        p.rshift_width = gt_right_shift_width;
39        _ctx->tpu_mul_const(&p);
40        bmk1880_min_const_param_t p1;
41        p1.min = neg;
42        p1.a = bottom;
43        p1.b = 0;
44        p1.b_is_signed = 1;
45        _ctx->tpu_min_const(&p1);
46        p.res_high = NULL;
47        p.res_low = neg;
48        p.a = neg;
49        p.b = le_scale;
50        p.b_is_signed = true;
51        p.rshift_width = le_right_shift_width;
52        _ctx->tpu_mul_const(&p);
53        bmk1880_or_int8_param_t p2;
54        p2.res = bottom;
55        p2.a = relu;
56        p2.b = neg;
57        _ctx->tpu_or_int8(&p2);
58        //move result to global
59        _ctx->gdma_store(bottom, slice_top_gaddr, CTRL_NEURON);
60        //free
61        _ctx->tl_free(neg);
62        _ctx->tl_free(relu);
63        _ctx->tl_free(bottom);
64
65        slice_bottom_gaddr += count_sec * INT8_SIZE;
66        slice_top_gaddr += count_sec * INT8_SIZE;
67    }
68  }
```

6.3.2 CPU Version

If the IR could only be converted using CPU, please add a new cpu op, and store IR op_ to it:

```
1  void TGLeakyReluFixedInst::encode() {
2    op_.add_threshold_x(m_inputCalibrationParameter[0].blob_param(0).
       threshold_y());
```

```
  op_.add_threshold_y(m_calibrationParameter.blob_param(0).
      threshold_y());
  add_cpu_op(_ctx->bm_get_bmk(),"LeakyReluOp", op_);
}
```

Navigate to the cpu_op folder, and create a new cpp source file, the name of which should be same as type name of customized layer. In the file, you need to create a child class that inherited from CpuOp, and implement run() member method with c++ code. Finally, please register the new class with REGISTER_CPU_OP().

```
#include <bmnet/targets/cpu/cpu_op.hpp>

namespace bmnet {

class LeakyReluOp: public CpuOp {
public:
void run() {
    assert(op_.type() == "ELU");
    const TensorShape& input_shape = op_.input_shape(0);
    int count = GetTensorCount(input_shape);
    char *input_data = NULL;
    char *output_data = NULL;
    float *bottom_data = NULL;
    float *top_data = NULL;
    if (op_.threshold_x_size() > 0) {
      input_data = reinterpret_cast<char*>(op_.global_input(0));
      bottom_data = (float*)malloc(sizeof(float) * count);
      for (int i = 0; i < count; ++i) {
        bottom_data[i] = input_data[i] * op_.threshold_x(0) /128.0;
      }
    }
    else {
      bottom_data = reinterpret_cast<float*>(op_.global_input(0));
    }
    if (op_.threshold_y_size() > 0) {
      output_data = reinterpret_cast<char*>(op_.global_output(0));
      top_data = (float*)malloc(sizeof(float) * count);
    }
    else {
      top_data = reinterpret_cast<float*>(op_.global_output(0));
    }
    float negative_slope = op_.tg_customized_param().f32_param(0);
    for (int i = 0; i < count; ++i) {
```

```
34        top_data[i] = std::max(bottom_data[i], (float)0)
35            + negative_slope * (std::min(bottom_data[i], (float)0));
36      }
37      for (int i = 0; i < count; ++i) {
38        if (op_.threshold_y_size() > 0) {
39          int fixed_data = (int)(top_data[i] * 128  / op_.threshold_y
                (0) + 0.5);
40          output_data[i] = (fixed_data < -128) ? -128 : ((fixed_data
                > 127) ? 127 : fixed_data);
41        }
42      }
43      if (op_.threshold_y_size() > 0) {
44        free(top_data);
45      }
46      if (op_.threshold_x_size() > 0) {
47        free(bottom_data);
48      }
49    }
50  };
51  } //namespace of bmnet.
52  // register CPU OP LeakyReluOp
53  REGISTER_CPU_OP(LeakyReluOp);
```

In order to compile the new added source file, please add it the CMakeLists.txt in the same folder.

6.4 Add instances to builder.

Create instances of new added CAFFE layer and tensor instruction, and then register them to builder.

```
1  CaffeBuilder *builder = new CaffeBuilder(
2  arg.target, arg.modified_proto,
3      arg.caffemodel, arg.weight_bin);
4
5  builder->addCustomizedLayer(new LeakyReluLayer());
6  builder->addCustomizedTensorInst(new TGLeakyReluFixedInst());
7
8  builder->build(shapes[0], shapes[1], shapes[2], shapes[3], option);
```

7 API Sample Code

Sample code will show you how to implement the function of bm_builder.bin,make sure you network can computed with INT8.You can add options as you need to implement the function of these tool files.

7.1 bm_builder.bin

```
/*
* Copyright (C) Bitmain Technologies Inc.
* All Rights Reserved.
*/
#include <iostream>
#include <cstring>
#include <map>
#include <string>
#include <vector>
#include <fstream>
#include <getopt.h>

#include <bmkernel/bm_kernel.h>
#include <bmnet/CaffeBuilder.hpp>

using namespace::bmnet;

  int          target     = BM_CHIP_BM1880;
  const char *name        = "alexnet";
  const char *caffemodel = "/mybmnet/out/alexnet/bmnet_alexnet_int8
     .1x10.caffemodel";
  const char *shape       = "1,3,227,227";
  const char *plugin      = "/mybmnet/lib";
  const char *in_ctable   = "/mybmnet/out/alexnet/
     bmnet_alexnet_calibration_table.1x10.pb2";
  const char *out_model   = "alexnet_1_bmodel_int8.bin";
  const char *out_ctable = "alexnet_ctable_opt.pb2";
```

```
26
27  int main(int argc, char** argv) {
28    google::SetStderrLogging(google::GLOG_INFO);
29    google::InitGoogleLogging(argv[0]);
30
31    std::vector<int> shapes;
32    shapes.push_back(1);
33    shapes.push_back(3);
34    shapes.push_back(227);
35    shapes.push_back(227);
36
37    CaffeBuilder<int8_t> *builder =
38      new CaffeBuilder<int8_t>(target,NULL,caffemodel,NULL,
          in_ctable, out_ctable);
39
40    int option = 0;
41    option = BM_OPT_LAYER_GROUP_WITH_WEIGHT_OPT;
42
43    builder->build(shapes[0], shapes[1], shapes[2], shapes[3],
        option);
44
45    builder->store_model(name, out_model, plugin);
46
47    delete builder;
48    return 0;
49 }
```