NNToolChain Documentation

Sophon

Apr 12, 2019

CONTENTS

NNToolChain 是用于编译和部署深度学习模型到 Sophon TPU 的工具 (https://sophon.ai), 包含

1. Caffe/TensorFlow/MXNet/PyTorch 的前端 parser
2. 计算图优化的编译器及运行时
3. Layer-based API: bmdnn/bmcv (similar to cuDNN for NVIDIA GPU);
4. Tensor/Operator-based API: bmlang (writing custom op);

CHAPTER

ONE

NNTOOLCHAIN 快速开始

1.1 NNToolChain 基本概念介绍

BMNNSDK

比特大陆原创深度学习开发工具包

BM168x

比特大陆面向深度学习领域推出的第 x 代云端张量处理器

PCIE-Mode

一种产品形态，SDK 运行于 X86 平台，BM168x 作为 PCIE 接口的深度学习计算加速卡存在

SOC-Mode

一种产品形态，SDK 独立运行于 BM168x 平台，支持通过千兆以太网与其他设备互联

BMCompile

是一个面向比特大陆 TPU 处理器研发的深度神经网络的优化编译器，可以将深度学习框架定义的各种深度神经网络转化为 TPU 上运行的指令流。

BMRuntime: 匹配 BMCompiler 的运行时库，提供上层应用程序可编程调用的接口

BMDNN: 深度学习 layer 级别的加速库接口

BMCV: 使用 TPU 进行 CV 处理的加速库接口

BMNetC: 面向 Caffe model 的 BMCompiler 前端

BMNetT: 面向 TensorFlow model 的 BMCompiler 前端

BMNetM: 面向 MxNet model 的 BMCompiler 前端

BMNetP: 面向 PyTorch model 的 BMCompiler 前端

BMLang: 面向 TPU 的高级编程模型，用户开发时无需了解底层 TPU 硬件信息

bmodel: 面向比特大陆 TPU 处理器的深度神经网络模型文件格式

1.2 版本特性

BMNNSDK 包含设备驱动、运行库、头文件和相应工具，主要特性如下：

1. 设备驱动
   - PCIE 支持多种 Linux 发行版本和 Linux 内核。
   - SOC 模式提供 ko 模块，可以直接安装到 BM168x SOC Linux Release。
2. 运行库
   - 提供深度学习推理引擎，提供最大的推理吞吐量和最简单的应用部署环境；

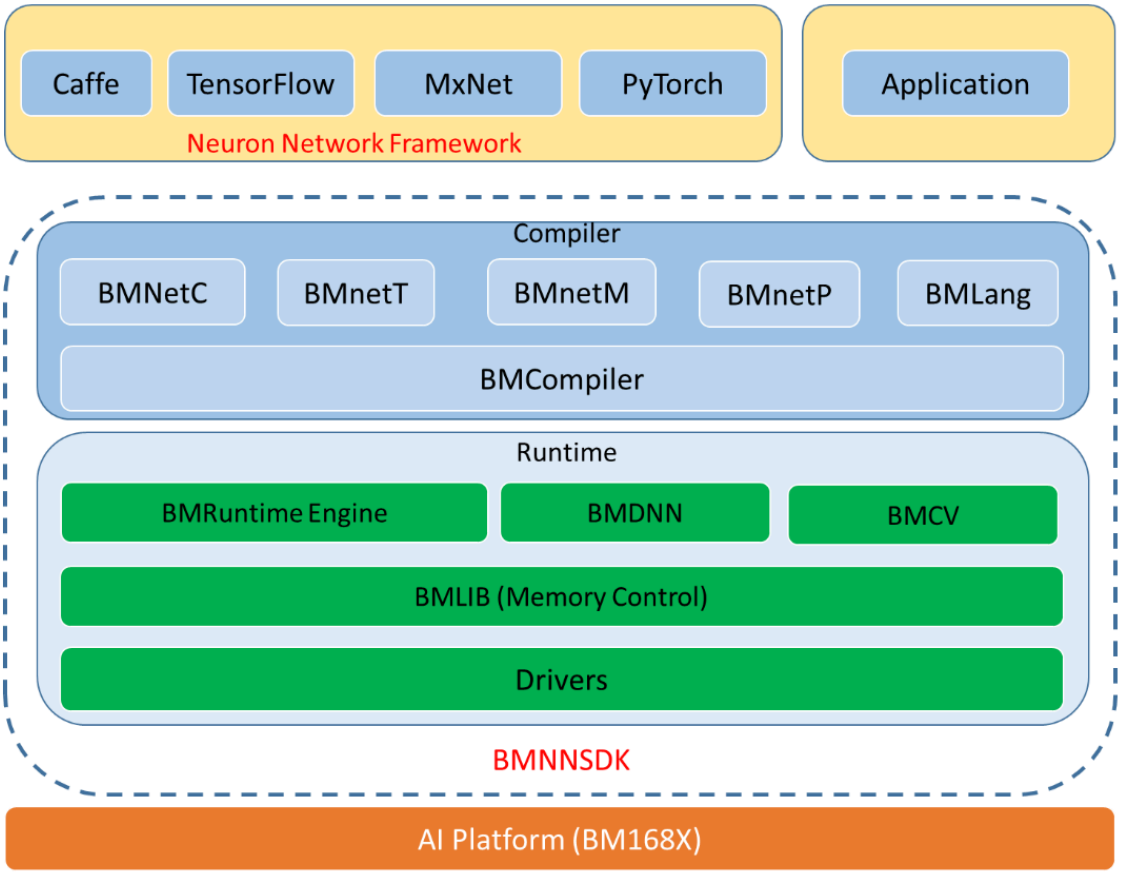
- 提供三层接口，网络级接口/layer 级接口/指令集接口
- 利用 TPU 提供灵活的图像处理功能 (BMCV)
- 提供运行库编程接口，用户可以直接操作 bmlib/bmdnn 等底层接口，进行深度的开发
- 运行库支持多线程、多进程，提供并发处理能力。

3. 工具
   - 提供 bmnetc 工具，支持 Caffe 网络模型进行编译
   - 提供 bmnett 工具, 支持 TensorFlow 模型进行编译
   - 提供 bmnetm 工具，支持 MxNet 模型进行编译
   - 提供 bmnetp 工具，支持 PyTorch 模型进行编译
   - 提供 bm_model.bin 工具，查看 bmodel 文件的参数信息，也可以将 bmodel 文件进行分解和合并
   - 提供 profiling 的工具，展示每一层执行所使用的指令和指令所消耗的时间
   - 提供 bm_smi 工具，支持对 BM1682 设备的运行状态进行监测

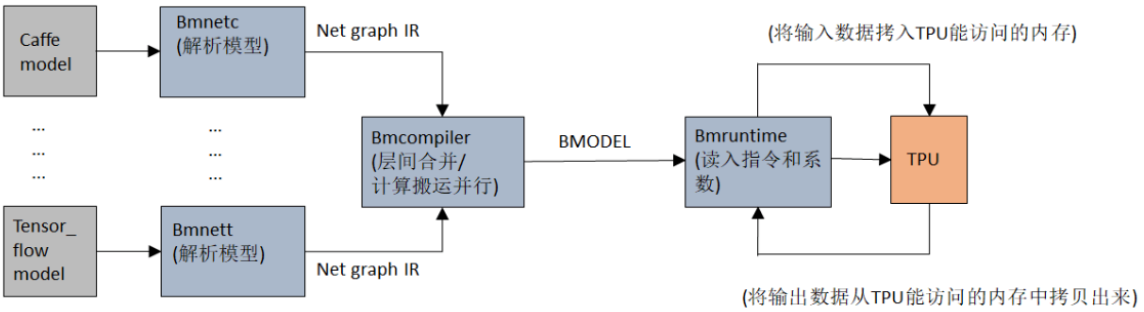
1.3 NNToolChain 整体架构

BMNNSDK（Bitmain Neural Network SDK）是比特大陆基于其自主研发的 AI 芯片，所定制的深度学习 SDK，涵盖了神经网络推理阶段所需的模型优化、高效运行时支持等能力，为深度学习应用开发和部署提供易用、高效的全栈式解决方案。

BMNNSDK 由 Compiler 和 Runtime 两部分组成，Compiler 负责对各种深度神经网络模型（如 caffemodel、tensorflow model 等）进行编译和优化，最终生成运行时需要的 Bmodel。Runtime 负责驱动 TPU 芯片，为上层应用程序提供统一的可编程接口，既能使程序可以通过 Compiler 编译的 Context 进行神经网络推理，也能使用 BMDNN 和 BMCV 对 DNN 和 CV 算法进行加速，而用户无需关心底层硬件实现细节。



Compiler 是一个模型转换工具，可以对各种框架的模型进行离线转换，将模型转换成 TPU 能够执行的模型格式，然后调用 bmrutime 在初始化阶段读取模型，运行时则将输入数据拷给 TPU，TPU 进行神经网络推理，再将输出读取出来。



1.4 NNToolChain 安装说明

针对不同的应用场景，BITMAIN 基于 BM168x 芯片提供了 PCIE 和 SOC 两种产品形态，对应于两种不同的运行模式：

- PCIE 模式：SDK 运行于 X86 平台，BM168x 作为 PCIE 接口的深度学习计算加速卡存在；
- SOC 模式：SDK 独立运行于 BM168x 平台，支持通过千兆以太网与其他设备连接。

BMNNSDK 支持以上两种运行模式，在应用接口上完全兼容，请根据产品形态和需求下载对应的 BMNNSDK 进行安装。

1.5 PCIE 模式安装

1. 安装需求
    1. 硬件环境：带有 PCIE 接口的 X86 主机
    2. 操作系统：Ubuntu/CentOS/Debian
    3. PCIE 模式的 BMDNNSDK 安装包
    4. 第三方依赖：
        - libopencv-dev
        - libboost-all-dev
        - libgflags-dev
2. 安装方式

   进入安装目录，执行以下脚本，其功能包括初始化库的路径以及添加一些编译或运行的示例函数等。

   ```
   cd release_dir source envsetup.sh
   ```

   安装驱动，等待结束后就安装成功了。

   ```
   sudo ./install_driver.sh
   ```

1.6 SoC 模式安装

1. 安装需求
    1. 硬件环境：BM168x SOC 板卡
    2. 操作系统：BM168x Linux Release
    3. 带有升级包的 TF 卡
2. 安装方式
    1. 板卡断电后，插入带有升级包的 TF 卡。
    2. 上电后会看到外壳 power 绿灯和 err 红灯交替闪烁，表明正在升级。
    3. 几分钟后闪烁结束，只有 power 绿灯亮，表示升级结束。
    4. 断电并拔出 TF 卡。
    5. 重新上电，bmdnn 的环境会从 eMMC 启动，BMDNNSDK 安装在了/system/bmdnn/目录下。

CHAPTER

TWO

NNTOOLCHAIN 使用说明

2.1 BMNETC 使用

2.1.1 编译 Caffe 模型

BMNETC 是针对 caffe 的模型编译器，可将某网络的 caffemodel 和 prototxt 编译成 BMRuntime 所需要的文件。而且在编译的同时，支持每一层的 NPU 模型计算结果都会和 CPU 的计算结果进行对比，保证正确性。下面介绍该编译器的使用方式。

1. 安装需求
   - python 3.5.x
   - linux
2. 配置步骤

   设置 LD_LIBRARY_PATH。可以使用以下方式在当前 shell 中设置该库的路径，或者也可以选择将该路径的设置永久地添加到‘.bashrc’文件中，如下：

   ```
   export LD_LIBRARY_PATH=path_to_bmcompiler_lib:$ LD_LIBRARY_PATH
   ```

3. 使用方法

   1. 方式一：命令形式

   Command name: bmnetc - BMNet compiler command for Caffe model

   ```
   /path/to/bmnetc [--model=<path>] \
                   [--weight=<path>] \
                   [--shapes=<string>] \
                   [--net_name=<name>] \
                   [--opt=<value>] \
                   [--dyn=<bool>] \
                   [--outdir=<path>] \
                   [--target=<name>] \
                   [--cmp=<bool>] \
                   [--mode=<string>]
   ```

   参数介绍如下：

   其中 mode 为 compile 表示编译 float 模型。为 GenUmodel 表示生成 BITMAIN 定义的统一 model，可后续通过 BITMAIN 定点化工具进行 INT8 定点化生成 INT8 model。

   GenUmodel 模式下，参数 opt、dyn、target、cmp 将没有意义，无需指定。

args	type	Description
model	string	**Necessary**. Caffe prototxt path
weight	string	**Necessary**. Caffemodel(weight) path
shapes	string	**Optional**. Shapes of all inputs, default use the shape in prototxt, format [x,x,x,x],[x,x]···, these correspond to inputs one by one in sequence
net_name	string	**Optional**. Name of the network, default use the name in prototxt
opt	int	**Optional**. Optimization level. Option: 0, 1, 2, default 2.
dyn	bool	**Optional**. Use dynamic compilation, default false.
outdir	string	**Necessary**. Output directory
target	string	**Necessary**. Option: BM1682, BM1684; default: BM1682
cmp	bool	**Optional**.Check result during compilation. Default: true
mode	string	**Optional**. Set bmnetc mode. Option: compile, GenUmodel. Default: compile.

examples:

以下是编译 float32 的 caffe model 命令示例

```
/path/to/bmnetc --model=/path/to/prototxt --weight=/path/to/caffemodel --shapes=[1,3,224,
↪224] --net_name=resnet18 --outdir=./resnet18 target=BM1682
```

以下是生成 Umodel 的示例

```
/path/to/bmnetc --mode=GenUmodel --model=/path/to/prototxt --weight=/path/to/caffemodel --
↪shapes=[1,3,224,224] --net_name=resnet18 --outdir=./resnet18
```

2. 方式二：python 接口

bmnetc 的 python 接口如下, 需要 pip3 install –user bmnetc-x.x.x-py2.py3-none-any.whl。

以下是编译 float32 的 caffe model 的 python 接口。

```
import bmnetc
## compile fp32 model
bmnetc.compile(
  model = "/path/to/prototxt",     ## Necessary
  weight = "/path/to/caffemodel", ## Necessary
  ourdir = "xxx",                  ## Necessary
  target = "BM1682",               ## Necessary
  shapes = [[x,x,x,x], [x,x,x]],  ## optional, if not set, default use shape in prototxt
  net_name = "name",               ## optional, if not set, default use the network name␣
↪in prototxt
  opt = 2,                         ## optional, if not set, default equal to 2
  dyn = False,                     ## optional, if not set, default equal to False
  cmp = True                       ## optional, if not set, default equal to True
)
```

以下是生成 Umodel 的 python 接口。

```
import bmnetc
## Generate BITMAIN U model
bmnetc.GenUmodel(
  model = "/path/to/prototxt",     ## Necessary
  weight = "/path/to/caffemodel", ## Necessary
  ourdir = "xxx",                  ## Necessary
  shapes = [[x,x,x,x], [x,x,x]],  ## optional, if not set, default use shape in prototxt
  net_name = "name"                ## optional, if not set, default use the network name␣
↪in prototxt
)
```

以下是使用 bmnetc python 的例子:

```
import bmnetc

model = r'../../../nnmodel/caffe_models/lenet/lenet_train_test_thin_4.prototxt'
weight = r'../../../nnmodel/caffe_models/lenet/lenet_thin_iter_1000.caffemodel'
target = r"BM1682"
export_dir = r"./compilation"
shapes = [[1,1,28,28],[1]]

bmnetc.compile(model = model, weight = weight, target = target, outdir = export_dir,␣
↪shapes = shapes)
bmnetc.GenUmodel(weight = weight, model = model, net_name = "lenet")
```

2.1.2 实现 Caffe 自定义 Layer

BMNetC 前端下自定义 layer 实现的可编程环境为 imp_bmnetc，该编程环境与 Caffe 相同。

基于 BMLang 开发

BMLang 是提供用户针对 BMTPU 编程的接口，所实现的算法可以在 BMTPU 中运行，详细见 *BMLang 说明* 。

这里介绍如何在 bmnetc 编程环境下对自定义 Layer 和 bmnetc 未支持的 layer 进行 TPU 编程，并将 BMLang 实现的 layer 插入到网络中，与其他 layer 一起进行网络级的编译，生成网络的 bmodel。

以下是以 exp layer 为例介绍基于 BMLang 开发网络中未支持 layer 的步骤:

1. 修改 prototxt 文件

首先需要修改 prototxt 中 bmnetc 不支持的 layer type 的 param。bmnetc 提供给用户自定义 layer 的 google proto parameters 格式如下:

```
message UserDefinedParameter {
  repeated float float_value = 1;
  repeated string string_value = 2;
}
```

修改.prototxt 里面的 type 为 Exp 的 layer param。caffe 原版的 Exp prototxt 格式为:

```
layer {
  name: "exp"
  type: "Exp"
  bottom: "log"
  top: "usr"
  exp_param {
    base: 2
    scale: 2
    shift: 3
  }
}
```

需要修改成:

```
layer {
  name: "exp"
  type: "Exp"
  bottom: "log"
  top: "usr"
  user_defined_param {
    float_value: 2
    float_value: 2
```

(continues on next page)

(continued from previous page)

```
    float_value: 3
  }
}
```

2. 然后在 imp_bmnetc 的 inclue 和 src 里，加入 exp layer 的.hpp 和.cpp 代码文件。

   示例.hpp 代码如下：

```
#ifndef CAFFE_USER_DEFINED_LAYER_HPP_
#define CAFFE_USER_DEFINED_LAYER_HPP_

#include <vector>

#include "bmnetc/blob.hpp"
#include "bmnetc/layer.hpp"
#include "bmnetc/proto/bmnetc.pb.h"

#include "bmnetc/layers/neuron_layer.hpp"

namespace bmnetc {

/**
 * @brief Computes @f$ y = \gamma ^ {\alpha x + \beta} @f$,
 *        as specified by the scale @f$ \alpha @f$, shift @f$ \beta @f$,
 *        and base @f$ \gamma @f$.
 */
template <typename Dtype>
class ExpLayer : public NeuronLayer<Dtype> {
 public:
  /**
   * @param param provides UserDefinedParameter UserDefined_param,
   *     with ExpLayer options:
   */
  explicit ExpLayer(const LayerParameter& param)
      : NeuronLayer<Dtype>(param) {}
  virtual void LayerSetUp(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top);

  virtual inline const char* type() const { return "Exp"; }

 protected:
  /**
   * @param bottom input Blob vector (length 1)
   *   -# @f$ (N \times C \times H \times W) @f$
   *      the inputs @f$ x @f$
   * @param top output Blob vector (length 1)
   *   -# @f$ (N \times C \times H \times W) @f$
   *      the computed outputs @f$
   *        y = \gamma ^ {\alpha x + \beta}
   *      @f$
   */
  virtual void CheckBlobCounts(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top){};
  virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top);
  virtual void layer_deploy(void* p_bmcompiler, const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top);
  virtual void Forward_bmtpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top);

  void bmtpu_module(const vector<Blob<Dtype>*>& bottom,
```

(continues on next page)

(continued from previous page)

```
    const vector<Blob<Dtype>*>& top);

  Dtype inner_scale_, outer_scale_;
};

}  // namespace bmnetc

#endif
```

示例.cpp 代码如下:

其中，LayerSetup 需要获取的是 UserDefinedParameter，然后根据该 UserDefinedParameter 设置 Exp Layer 的变量。

bmtpu_module 是用 BMLang 对 Exp 编程的模块。

Forward_bmtpu 中，需要 set_mode(BMLANG_COMPUTE)，表明使用 cpu 来模拟 bmtpu_module 的计算，以便于我们调试用 BMLang 对 exp layer 编程对不对。

layer_deploy 则需要 set_mode(BMLANG_COMPILE)，此时进入 compile 模式。在运行 bmnetc 编译 caffe model 时，会进入 layer_deploy 函数，并生成可在 BMTPU 芯片运行的 bmodel。

```
#include <vector>

#include "exp_layer.hpp"
#include "bmnetc/util/math_functions.hpp"

namespace bmnetc {

template <typename Dtype>
void ExpLayer<Dtype>::LayerSetUp(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {
  NeuronLayer<Dtype>::LayerSetUp(bottom, top);
  const UserDefinedParameter& param = this->layer_param_.user_defined_param();
  const Dtype base = param.float_value(0);
  if (base != Dtype(-1)) {
    CHECK_GT(base, 0) << "base must be strictly positive.";
  }
  // If base == -1, interpret the base as e and set log_base = 1 exactly.
  // Otherwise, calculate its log UserDefinedlicitly.
  const Dtype log_base = (base == Dtype(-1)) ? Dtype(1) : log(base);
  CHECK(!isnan(log_base))
      << "NaN result: log(base) = log(" << base << ") = " << log_base;
  CHECK(!isinf(log_base))
      << "Inf result: log(base) = log(" << base << ") = " << log_base;
  const Dtype input_scale = param.float_value(1);
  const Dtype input_shift = param.float_value(2);
  inner_scale_ = log_base * input_scale;
  outer_scale_ = (input_shift == Dtype(0)) ? Dtype(1) :
     ( (base != Dtype(-1)) ? pow(base, input_shift) : exp(input_shift) );
}

template <typename Dtype>
void ExpLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
  const int count = bottom[0]->count();
  const Dtype* bottom_data = bottom[0]->cpu_data();
  Dtype* top_data = top[0]->mutable_cpu_data();
  if (inner_scale_ == Dtype(1)) {
    bmnetc_exp(count, bottom_data, top_data);
  } else {
    bmnetc_cpu_scale(count, inner_scale_, bottom_data, top_data);
```

(continues on next page)

(continued from previous page)

```
    bmnetc_exp(count, top_data, top_data);
  }
  if (outer_scale_ != Dtype(1)) {
    bmnetc_scal(count, outer_scale_, top_data);
  }
}

template <typename Dtype>
void ExpLayer<Dtype>::layer_deploy(void* p_bmcompiler,
        const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top)
{
#if defined(BMCOMPILE) && defined(BMLANG)
  set_mode(BMLANG_COMPILE);
  bmtpu_module(bottom,top);
#endif
}

template <typename Dtype>
void ExpLayer<Dtype>::Forward_bmtpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
#if defined(BMCOMPILE) && defined(BMLANG)
  set_mode(BMLANG_COMPUTE);
  bmtpu_module(bottom,top);
#endif
}

template <typename Dtype>
void ExpLayer<Dtype>::bmtpu_module(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
#if defined(BMCOMPILE) && defined(BMLANG)
    bmtensor<Dtype> bottom_tensor(this->layer_param_.bottom(0),bottom[0]->shape());
    bottom_tensor.set_data(bottom[0]->cpu_data());
    bmtensor<Dtype> top_tensor(this->layer_param_.top(0),top[0]->shape());
    top_tensor.set_data(top[0]->cpu_data());

    bmtensor<Dtype> coeff_tensor("coef",1,1,1,1);
    coeff_tensor.set_tensor_type(COEFF_TENSOR);
    coeff_tensor.data()[0] = inner_scale_;
    bmtensor<Dtype> mul_tensor("mul_res");
    if (inner_scale_ != Dtype(1)) {
      bm_mul(bottom_tensor,coeff_tensor,mul_tensor);
    }else{
      bm_setdata(bottom_tensor,top_tensor);
    }

    bm_active(mul_tensor,top_tensor,ACTIVE_EXP);

    coeff_tensor.data()[0] = outer_scale_;
    if (outer_scale_ != Dtype(1)) {
      bm_mul(top_tensor,coeff_tensor,top_tensor);
    }
#endif
}

INSTANTIATE_CLASS(ExpLayer);
REGISTER_LAYER_CLASS(Exp);

}  // namespace bmnetc
```

3. 完成代码后，在 imp_bmnetc 下 source build.sh 即可编译，将生成的 libimpbmnetc.so 通过以下命令加入到 LD_LIBRARY_PATH 中。

```
export LD_LIBRARY_PATH=path_to_libimpbmnetc:$ LD_LIBRARY_PATH
```

4. 启动 bmnetc 编译该 caffe model，其中 prototxt 为修改后的，.caffemodel 为已经用原版.prototxt 训练好的。最终生成 bmodel，在 bmruntime 时可将 bmodel 下发到 BMTPU 芯片中运行。
5. 我们也可以对 BMLang 程序进行单元测试并调试，看 bmlang 描述的计算模式对不对。可以写一个 test 程序，创建该 ExpLayer，设置参数，启动 Forward_cpu()，再启动 Forward_tpu()，对比两个的结果是否相同。如果不同，则对 bmtpu_module 中的程序进行调试并修改。

2.2 BMNETT 使用

BMNETT 是针对 tensorflow 的模型编译器，在该 device 下创建 graph，可以被编译成 BMRuntime 所需的文件。而且在编译 graph 的同时，可选择将每一个操作的 NPU 模型计算结果和 CPU 的计算结果进行对比，保证正确性。下面分别介绍该编译器的安装需求、安装步骤和使用方法。

1. 安装需求
   - python 2.7.x
   - tensorflow>1.9.0
   - linux
2. 安装步骤

安装该编译器的安装包。选择以下命令其一使用。使用命令（1）则安装在本地目录中，不需要 root 权限，而命令（2）则安装在系统目录中，需要 root 权限。

```
pip install --user bmnett-x.x.x-py2.py3-none-any.whl
```

```
pip install bmnett-x.x.x-py2.py3-none-any.whl
```

设置 LD_LIBRARY_PATH。可以使用以下方式在当前 shell 中设置该库的路径，或者也可以选择将该路径的设置永久地添加到.bashrc 文件中。

```
export LD_LIBRARY_PATH=path_to_bmcompiler_lib
```

3. 使用方法
   1. 方式一：命令形式

   Command name: python -m bmnett - BMNet compiler command for TensorFlow model

```
python -m bmnett  [--model=<path>] \
                  [--input_names=<string>] \
                  [--shapes=<string>] \
                  [--output_names=<string>] \
                  [--net_name=<name>] \
                  [--opt=<value>] \
                  [--dyn=<bool>] \
                  [--outdir=<path>] \
                  [--target=<name>] \
                  [--cmp=<bool>] \
                  [--mode=<string>]
```

   参数介绍如下：

   其中 mode 为 compile 表示编译 float 模型。为 GenUmodel 表示生成 BITMAIN 定义的统一 model，可后续通过 BITMAIN 定点化工具进行 INT8 定点化生成 INT8 model。为 summay 或者 show 模式表示显示网络 graph。

   GenUmodel 模式下，参数 opt、dyn、target、cmp 将没有意义，无需指定。

args	type	Description
model	string	**Necessary**. Tensorflow .pb path
input_names	string	**Necessary**. Set name of all network inputs one by one in sequence. Format "name1,name2,name3"
shapes	string	**Necessary**. Shapes of all inputs, default use the shape in prototxt, format [x,x,x,x],[x,x]···, these correspond to inputs one by one in sequence
output_names	string	**Necessary**. Set name of all network outputs one by one in sequence. Format "name1,name2,name2"
net_name	string	**Necessary**. Name of the network, default use the name in prototxt
opt	int	**Optional**. Optimization level. Option: 0, 1, 2, default 1.
dyn	bool	**Optional**. Use dynamic compilation, default false.
outdir	string	**Necessary**. Output directory
target	string	**Necessary**. Option: BM1682, BM1684; default: BM1682
cmp	bool	**Optional**.Check result during compilation. Default: true
mode	string	**Optional**. Set bmnetc mode. Option: compile, GenUmodel, summary, show. Default: compile.

2. 方式二：python 接口

bmnett 编译 float32 tf model 的 python 接口如下：

```
import bmnett
## compile fp32 model
bmnett.compile(
  model = "/path/to/.pb",          ## Necessary
  ourdir = "xxx",                  ## Necessary
  target = "BM1682"                ## Necessary
  shapes = [[x,x,x,x], [x,x,x]],   ## Necessary
  net_name = "name",               ## Necessary
  input_names=["name1", "name2"], ## Necessary
  output_names=["out_name1", "out_name2"], ## Necessary
  opt = 2,                         ## optional, if not set, default equal to 1
  dyn = False,                     ## optional, if not set, default equal to False
  cmp = True                       ## optional, if not set, default equal to True
)
```

bmnett 转化 float32 tf model 为 BITMAIN U Model 的 python 接口如下：

```
import bmnett
## compile fp32 model
bmnett.GenUmodel(
  model = "/path/to/.pb",          ## Necessary
  ourdir = "xxx",                  ## Necessary
  shapes = [[x,x,x,x], [x,x,x]],   ## Necessary
  net_name = "name",               ## Necessary
  input_names=["name1", "name2"], ## Necessary
  output_names=["out_name1", "out_name2"] ## Necessary
)
```

2.3 BMNETM 使用

2.3.1 编译 MxNet 模型

BMNETM 是针对 mxnet 的模型编译器，可以将 mxnet 格式的模型结构文件和参数文件（比如：lenet-symbol.json 和 lenet-0100.params）在经过图编译优化后，转换成 BMRuntime 所需的文件。可选择将每一个操作的 NPU 模型计算结果和原始模型在 mxnet 框架上的计算结果进行对比，保证模型转换的正确性。下面分别介绍该编译器的安装需求、配置步骤和使用方法及命令参数简介。

1. 安装需求
   - python 3.5.x
   - mxnet>=1.3.0
   - linux
2. 配置步骤
   - 步骤 1：

```
# 安装 mxnet 1.3.0
pip3 install mxnet==1.3.0
# 验证 mxnet 安装的正确性，安装异常提示 No module named "mxnet";
python3 -c "import mxnet as mx"
```

   - 步骤 2:

```
# 安装 bmnetm python 包
pip3 install --user bmnetm-x.x.x-py2.py3-none-any.whl
```

   - 步骤 3:

   设置 LD_LIBRARY_PATH。可以使用以下方式在当前 shell 中设置该库的路径，或者也可以选择将该路径的设置永久地添加到‘.bashrc’文件中，如下：

   ```
   export LD_LIBRARY_PATH=path_to_bmcompiler_lib:$ LD_LIBRARY_PATH
   ```

3. 使用方法

   1. 方式一：命令形式

   Command name: python3 -m bmnetm - BMNet compiler command for MxNet model

```
python3 -m bmnetm [--model=<path>] \
                  [--weight=<path>] \
                  [--shapes=<string>] \
                  [--input_names=<string>] \
                  [--net_name=<name>] \
                  [--opt=<value>] \
                  [--dyn=<bool>] \
                  [--outdir=<path>] \
                  [--target=<name>] \
                  [--cmp=<bool>]
```

   参数介绍如下：

args	type	Description
model	string	**Necessary**. MxNet symbol .json path
weight	string	**Necessary**. MxNet weight .params path
shapes	string	**Necessary**. Shapes of all inputs, default use the shape in prototxt, format [x,x,x,x],[x,x]···, these correspond to inputs one by one in sequence
input_names	string	**Optional**. Set input name according to .json. They correspond to shapes one by one. Default: “data” . Format “name1,name2,···” .
net_name	string	**Necessary**. Name of the network, default use the name in prototxt
opt	int	**Optional**. Optimization level. Option: 0, 1, 2, default 1.
dyn	bool	**Optional**. Use dynamic compilation, default false.
outdir	string	**Necessary**. Output directory
target	string	**Necessary**. Option: BM1682, BM1684; default: BM1682
cmp	bool	**Optional**.Check result during compilation. Default: true

   2. 方式二：python 接口

   bmnetm 的 python 接口如下：

```
import bmnetm
## compile fp32 model
bmnetm.compile(
  model = "/path/to/.json",        ## Necessary
  weight = "/path/to/.params",     ## Necessary
  ourdir = "xxx",                  ## Necessary
  target = "BM1682",               ## Necessary
  shapes = [[x,x,x,x], [x,x,x]],   ## Necessary
  net_name = "name",               ## Necessary
  input_names=["name1","name2"]    ## optional, if not set, default is "data"
  opt = 2,                         ## optional, if not set, default equal to 1
  dyn = False,                     ## optional, if not set, default equal to False
  cmp = True                       ## optional, if not set, default equal to True
)
```

2.3.2 实现 MxNet 自定义 layer

BMNetM 前端下自定义 layer 实现的可编程环境为 python 环境，该环境请见 layers_ext 文件夹。Note：bmnetm 下自定义 layer 加入网络一起编译目前支持使用 Python 接口的方式。

基于 BMLang 开发

BMLang 是提供用户针对 BMTPU 编程的接口，所实现的算法可以在 BMTPU 中运行，详细见 *BMLang 说明* 。

这里介绍如何在 bmnetm 编程环境下对自定义 layer 或者 bmnetm 未支持的 layer 进行 TPU 编程，并将 BMLang 实现的 layer 插入到网络中，与其他 layer 一起进行网络级的编译，生成网络的 bmodel。

这里我们以 activate_layer 为例，介绍基于 BMLang 开发网络中未支持 layer 的步骤：

1. MxNet 的 python 包更新

在进行 BMLang 开发前，需要将包含自定义 layer 实现的 mxnet python 包安装替换原来的 mxnet python 包。如果已经安装的 mxnet python 包中已经包含用户自定义 layer 的 cpu 实现，则直接进入第 2 步。

2. 在 layer_ext 文件夹里加入代码文件

文件夹中已经存在 activate_layer.py 这个 example。其源代码为：

其中，activation_core 函数是使用 BMLang 对 activate 这个算法进行编程。user_activation 则是解析改 OP 的参数，并调用 activation_core。

```
'''
The following is an example that use bmlang(python) to implement a operation for␣
↪BITMAIN TPU
A typical bmlang program for an OP implementation should include
1. A compute core that is written by bmlang
2. A register that import the bmlang program to bmnetm compiler
3. (Optional) A debug module that check the accuracy of bmlang program
'''

import bmlang

ACTIVE_TYPE_DICT = {
    'tanh'    : 0,
    'sigmoid' : 1,
    'Sigmoid' : 1,
    'relu'    : 2,
    'exp'     : 3,
    'elu'     : 4,
```

(continues on next page)

(continued from previous page)

```
    'sort'    : 5,
    'square'  : 6,
    'rsort'   : 7,
    'absval'  : 8,
    'ln'      : 9,
}

'''
The following is the activation compute core that is written by bmlang API
The parameters in this function are defined by users
'''
def activation_core(bottom_name, top_name, shape_dim, tensor_shape, active_type_id):
  print(shape_dim)
  print('in tensor name: ', bottom_name[0], 'out tensor name: ', top_name[0])
  if shape_dim == [4]:
    print('add active layer, dim is 4')
    ## create bmlang tensor of activation input
    inp_tensor = bmlang.bmtensor_float(bottom_name[0], tensor_shape[0][0], tensor_
↪shape[0][1], tensor_shape[0][2], tensor_shape[0][3])
    ## create bmlang tensor of activation output
    oup_tensor = bmlang.bmtensor_float(top_name[0], tensor_shape[0][0], tensor_
↪shape[0][1], tensor_shape[0][2], tensor_shape[0][3])
    ## bmlang compution operation
    bmlang.bm_active_float(inp_tensor, oup_tensor, active_type_id)
  elif shape_dim == [2]:
    print('add active layer, dim is 2')
    ## create bmlang tensor of activation input
    inp_tensor = bmlang.bmtensor_float(bottom_name[0], tensor_shape[0][0], tensor_
↪shape[0][1])
    ## create bmlang tensor of activation output
    oup_tensor = bmlang.bmtensor_float(top_name[0], tensor_shape[0][0], tensor_
↪shape[0][1])
    ## bmlang compution operation
    bmlang.bm_active_float(inp_tensor, oup_tensor, active_type_id)

'''
The following is the register that import bmlang program to bmnetm compiler
When we finish it. We must register this function in layer_ext_register.py
The name of the function can be defined by users
But the paramteters must be set as follows
  @param node           The node of mxnet. It correspond to each node of mxnet␣
↪symbol.json
  @param tensor         The input and output tensors of the node
  The tensor information include
    tensor.InTensorName           The name of the inputs of this node. It is␣
↪corresponding to the input name in mxnet symbol.json
    tensor.InTensorShape          The input shapes of the node.
    tensor.InTensorDim            The dimension number of each input shapes.
    tensor.InTensorRawData        The raw data of tensor. Mainly for coefficient.
    tensor.OutTensorName          The name of the outputs of this node. It is␣
↪corresponding to the output name in mxnet symbol.json
    tensor.OutTensorShape         The output shapes of the node.
    tensor.OutTensorDim           The dimension number of each output shapes.
'''
def user_activation(node, tensor):
  '''
  In the register, we should firstly parse the params of the OP
  The following is parser of this OP
  The key/value of node is the same with node information in symbol.json
  '''
  if (ACTIVE_TYPE_DICT.__contains__(node["op"])):
```

(continues on next page)

(continued from previous page)

```
    active_type_id = ACTIVE_TYPE_DICT.get(node["op"])
  elif (ACTIVE_TYPE_DICT.__contains__(node["attrs"]["act_type"])):
    active_type_id = ACTIVE_TYPE_DICT.get(node["attrs"]["act_type"])
  elif (ACTIVE_TYPE_DICT.__contains__(node["param"]["act_type"])):
    active_type_id = ACTIVE_TYPE_DICT.get(node["param"]["act_type"])
  else:
    raise RuntimeError("Not support activate layer type")

  print('EXT Factory: Bmlang activation is called')

  ## Get information of the input and output tensor
  shape_dim = tensor.InTensorDim
  bottom_name = tensor.InTensorName
  top_name = tensor.OutTensorName
  tensor_shape = tensor.InTensorShape

  ## Then set the bmlang model to COMPILE
  ## Now we set the mode of bmlang to BMLANG_COMPILE
  ## This mean compile this OP through bmcompiler
  bmlang.set_mode(bmlang.BMLANG_COMPILE)
  ## At last, call the compute core written through bmlang
  ## Compile the activation core
  activation_core(bottom_name, top_name, shape_dim, tensor_shape, active_type_id)

'''
BMLang debug example
'''
def bmlang_active_debug():
  ## 1. call mxnet active cpu compuation
  ## 2. bmlang.set_mode(bmlang.BMLANG_COMPILE)
  ## 3. call activation_core()
  ## 4. compare results from 1 and 3 above
```

3. 在 layer_ext_register.py 代码中给 bmnetm 注册 user_activation 这个函数

   代码如下，如果要注册其它的自定义 layer 可以类似 Activation 一样在这里增加注册。

```
import bmnetm
from layers_ext.activate_layer import *

## The layer that is wrote by bmlang can be registered here
def layer_ext_register():
    ## bmnetm.register('node_op_name', bmlang_register_func)
    bmnetm.register('Activation', user_activation)
```

   其中’Activation’为该 OP，对应.json 文件中的”op”： “Activation”

```
{
  "op": "Activation",
  "name": "relu4",
  "attrs": {"act_type": "relu"},
  "inputs": [[19, 0, 0]]
}
```

4. 基于 Python 来编译 MxNet 模型

   以下是以 lenet 为例来介绍，在使用 bmnetm compile 接口前，需要执行之前的注册程序 layer_ext_register()。

```
#model = r'/path/to/xxx-symbol.json'
model = r'../../../nnmodel/mxnet_models/lenet/lenet-symbol.json'
```

(continues on next page)

(continued from previous page)

```
#weight = r'/path/to/xxxx-xxxx.params'
weight = r'../../../nnmodel/mxnet_models/lenet/lenet-0100.params'

#export_dir = r'./xxx'
export_dir = r'./compilation'

#set target
target = r'BM1682'

#set input shapes
shapes = [[1, 1, 28, 28]]

#set network name
net_name = r'lenet'

## If user writes user-defined layers through bmlang, please register these layers␣
↪firstly
## If user does not have user-defined layer through bmlang, please delete the␣
↪following
import layers_ext.layer_ext_register as new_register
new_register.layer_ext_register()

## Launch bmnetm compilation
import bmnetm
bmnetm.compile(model, weight, export_dir, target, shapes, net_name)
```

5. 运行 Python 进行模型编译，最终将生成.bmodel，之后可用 BMRuntime 接口驱动.bmodel 在 BMTPU 芯片上运行。假设上述程序文件是 example.py，则：

   ```
   python3 example.py
   ```

   若仍有为支持的 layer，程序会报错提示。然后开发新 layer BMLang 实现。

6. 我们也可以对 BMLang 程序进行单元测试并调试，看 bmlang 描述的计算模式对不对。可以写一个 test 程序，调用 mxnet 的 activation 计算，在使用 bmlang.set_mode(bmlang.BMLANG_COMPUTE) 后调用 activation_core，对两个输出结果进行比对。若不正确，则调试修改 activation_core 程序。

2.4 BMNETP 使用

BMNETP 是针对 pytorch 的模型编译器，可以把 pytorch 的 model 直接编译成 BMRuntime 所需的执行指令。支持 python 代码或已经保存的 trace 文件。在编译 model 的同时，可选择将每一个操作的 NPU 模型计算结果和 CPU 的计算结果进行对比，保证正确性。下面分别介绍该编译器的安装需求、安装步骤和使用方法。

1. 安装需求

   - python 3.5.x
   - linux
   - pytorch==1.0.0、torchvision

2. 安装步骤

安装该编译器的安装包。选择以下命令其一使用。使用命令（1）则安装在本地目录中，不需要 root 权限，而命令（2）则安装在系统目录中，需要 root 权限。

```
pip install --user bmnetp-x.x.x-py2.py3-none-any.whl
```

```
pip install bmnetp-x.x.x-py2.py3-none-any.whl
```

3. 使用方法

1. 方式一：命令形式

Command name: python3 -m bmnetp - BMNet compiler command for PyTorch model

```
python3 -m bmnetp [--model=<path>] \
                  [--shapes=<string>] \
                  [--net_name=<name>] \
                  [--opt=<value>] \
                  [--dyn=<bool>] \
                  [--outdir=<path>] \
                  [--target=<name>] \
                  [--cmp=<bool>]
```

参数介绍如下：

args	type	Description
model	string	**Necessary**. PyTorch model .pt/pth path
shapes	string	**Necessary**. Shapes of all inputs, default use the shape in prototxt, format [x,x,x,x],[x,x]···, these correspond to inputs one by one in sequence
net_name	string	**Necessary**. Name of the network, default use the name in prototxt
opt	int	**Optional**. Optimization level. Option: 0, 1, 2, default 1.
dyn	bool	**Optional**. Use dynamic compilation, default false.
outdir	string	**Necessary**. Output directory
target	string	**Necessary**. Option: BM1682, BM1684; default: BM1682
cmp	bool	**Optional**. Check result during compilation. Default: true
desc	string	**Optional**. Describe data type and value range of some input in format: "[index, data format, lower bound, upper bound]" , where data format could be fp32, int64. For example, [0, int64, 0, 100], meaning input of index 0 has data type as int64 and values in [0, 100). If no description of some input given, the data type will be fp32 as default and uniformly distributed in 0 ~ 1.

2. 方式二：python 接口

bmnetp 的 python 接口如下：

```
import bmnetp
## compile fp32 model
bmnetp.compile(
  model = "/path/to/.pth",          ## Necessary
  ourdir = "xxx",                   ## Necessary
  target = "BM1682"                 ## Necessary
  shapes = [[x,x,x,x], [x,x,x]],    ## Necessary
  net_name = "name",                ## Necessary
  opt = 2,                          ## optional, if not set, default equal to 1
  dyn = False,                      ## optional, if not set, default equal to False
  cmp = True                        ## optional, if not set, default equal to True
)
```

2.5 BMLang 说明

BMLANG 是在 BMTPU 的基础之上封装了一些基本操作，开放给客户，客户可以基于此来实现其自定义或现有平台不支持的一些 layer。BMLANG 现在支持的一些操作以及说明在下面第一章中详细讲述。对于 mul、div、add、sub、max、min、gt、ge、lt、le、eq、ne，tensorA, tensorB 可以为 data 或 coeff tensor，但不能同时为 coeff tensor。除了个别 ops 不能由输入 tensor 推断出输出 tensor shape，其它 ops 由推断输出 tensor shape 的功能。例外的 ops 为 bm_setdata。

2.5.1 Install

如果使用 C++ 编程，则需要链接 libbmlang_C11ABI0.so 或者 libbmlang_C11ABI1.so，头文件为 bmlang_common.h，bmlang.h 和 bmtensor.h。

如果使用 Python 编程，需要安装 BMLang Python 包。

Python 版本为 2.x 需要安装

```
# 安装 bmlang python 2.x 包
pip install --user bmlang-x.x.x-py2-none-any.whl
```

Python 版本为 3.x 需要安装

```
# 安装 bmlang python 3.x 包
pip3 install --user bmlang-x.x.x-py2-none-any.whl
```

2.5.2 1. Class

C++ Class

bmtensor

简介：在 bmlang C++ API 中使用 bmtensor 来定义 tensor，支持 float 和 double 的数据类型，其具体的成员变量和方法的定义在 bmtensor.h 中。示例：定义一个 float 类型的 4D tensor，

```
bmtensor<float>  tensor_float("name", 2, 3, 4, 5);
```

Python Class

bmtensor_float

简介：在 bmlang python API 中使用 bmtensor_float 来定义 tensor，仅支持 float 的数据类型，其成员方法和变量的定义与 bmtensor 一致。示例：定义一个 4D 的 tensor，

```
Ftensor = bmtensor_float('name', 2, 3, 4, 5)
```

intVector

简介：python 中定义 vector<int> 类型的数据，其使用方法同 std::vector。示例：定义一个 vector 并赋值，

```
Intvec = intVector()
Intvec.push_back(6)
```

floatArray

简介：python 中定义一个 float 类型的数组。示例：定义并赋值一个数组，

```
Farray = floatArray(10)
for i in range(10):
  Farray[i] = i
```

bmtfPtrVector

简介：python 中定义一个类似 c++ 的 vector<bmtensor_float*> 示例：

```
refTenVec = bmtfPtrVector()
## Create other intermediate tensors
inpTenList = []
phTensor = bmtensor_float('x_tensor', 1, 10)
inpTenList.append(xTensor)
phTensor = bmtensor_float('h_tensor_t-1', 1, 10)
inpTenList.append(phTensor)
pcTensor = bmtensor_float('c_tensor_t-1', 1, 10)
inpTenList.append(pcTensor)
for j in range(len(inpTenList)):
  self.inpTenVec.push_back(inpTenList[j].this)
```

2.5.3 2 Operation Mode

2.1 mode

Bmlang 有两种工作模式：compute 和 compile。可以通过下面的接口实现：

```
set_mode(BMLANG_MODE_e mode);
typedef enum bmlang_mode {
  BMLANG_COMPUTE,
  BMLANG_COMPILE,
  BMLANG_BOTH,
  UNKNOWN_BMLANG_MODE
}BMLANG_MODE_e;
```

计算模式 (BMLANG_COMPUTE)：基于 cpu 实现 op，用来辅助比对基于 bmlang 实现的 layer 的正确性。编译模式 (BMLANG_COMPILE)：基于 BMTPU 实现 op，实现网络的编译。BMLANG_BOTH 则是既 COMPILE 又 COMPUTE，只有在调用 bmlang_compile_with_result_check 才是用这种 mode。

2.2 Standalone

Bmlang 可以脱离 bmnetc/bmnetp/bmnett/bmnetm 等编程框架使用。此时将单独实现算法在 BMTPU 里运行。以下接口用于单独使用 Bmlang。

1. 初始化 bmlang

   c++ API:

   ```
   void bmlang_init(const string& chip_name, BMLANG_MODE_e mode);
   ```

   python API:

   ```
   bmlang_init(chip_name, BMLANG_MODE_e)
   ```

2. 结束 bmlang

   c++ API:

   ```
   void bmlang_deinit();
   ```

   python API:

   ```
   bmlang_deinit()
   ```

3. 启动 compile only.

C++ API:

```
void bmlang_compile(const string& net_name, int opt_level, bool dyn_
↪cpl);
```

Pyhton API:

```
bmlang_compile(net_name, opt_level, dyn_cpl)
```

4. 启动 compile with result check。

C++ API:

```
void bmlang_compile_with_result_check(
    const string& net_name,
    int opt_level,
    bool dyn_cpl,
    vector<bmtensor<float>*> inp_tensor,
    vector<bmtensor<float>*> ref_tensor);
```

Python API:

```
bmlang_compile_with_result_check(
    net_name,
    opt_level,
    dyn_cpl,
    inp_tensor, ## it is bmtfPtrVector type
    ref_tensor) ## it is bmtfPtrVector type
```

2.5.4 3 Calculation Op

3.1 bm_mul

tensor_result = tensorA * tensorB，实现两个 tensor 按元素乘，支持数据类型：float32。

1. C++ API:

```
bm_mul(
const bmtensor<Dtype>& tensorA,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型。

2. Python API:

```
bm_mul_float(tensorA,  tensorB, tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型。

3.2 bm_div

tensor_result = tensorA / tensorB，实现两个 tensor 按元素除，支持数据类型：float32。

1. C++ API:

```
bm_div(
const bmtensor<Dtype>& tensorA,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型。

2. Python API:

```
bm_div_float(tensorA,  tensorB, tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型。

3.3 bm_add

tensor_result = tensorA + tensorB，实现两个 tensor 按元素加，支持数据类型：float32。

1. C++ API:

```
bm_add(
const bmtensor<Dtype>& tensorA,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型。

2. Python API:

```
bm_add_float(tensorA,  tensorB, tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型。

3.4 bm_sub

tensor_result = tensorA - tensorB，实现两个 tensor 按元素减，支持数据类型：float32。

1. C++ API:

```
bm_sub(
const bmtensor<Dtype>& tensorA,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型。

2. Python API:

```
bm_sub_float(tensorA,  tensorB, tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型。

3.5 bm_max

tensor_result = max(tensorA, tensorB)，实现两个 tensor 按元素取最大值，支持数据类型：float32。

1. C++ API:

```
bm_max(
const bmtensor<Dtype>& tensorA,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型。

2. Python API:

```
bm_max_float(tensorA,  tensorB, tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型。

3.6 bm_min

tensor_result = min(tensorA, tensorB)，实现两个 tensor 按元素取最小值，支持数据类型：float32。

1. C++ API:

```
bm_min(
const bmtensor<Dtype>& tensorA,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型。

2. Python API:

```
bm_min_float(tensorA,  tensorB, tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型。

3.7 bm_adds

tensor_result = tensorA + scalar，实现 tensorA 中每个元素都加上 scalar，支持数据类型：float32。

1. C++ API:

```
bm_adds(
const bmtensor<Dtype>& tensorA,
const Dtype scalar,
bmtensor<Dtype>& tensor_result)
```

参数说明：tensorA，tensor_result 均是 bmtensor 类型，scalar 是 float32 类型。

2. Python API:

```
bm_adds_float(tensorA,  scalar, tensor_result)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型，scalar 是标量类型。

3.8 bm_muls

tensor_result = tensorA * scalar，实现 tensorA 中每个元素都乘以 scalar，支持数据类型：float32。

1. C++ API:

```
bm_muls(
const bmtensor<Dtype>& tensorA,
const Dtype scalar,
bmtensor<Dtype>& tensor_result)
```

参数说明：tensorA，tensor_result 均是 bmtensor 类型，scalar 是 float32 类型。

2. Python API:

```
bm_muls_float(tensorA,  scalar, tensor_result)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型，scalar 是标量类型。

3.9 bm_maxs（TODO）

tensor_result = max(tensorA, scalar)，实现 tensorA 中每个元素都与 scalar 比较大小，取大值输出，支持数据类型：float32。

1. C++ API:

```
bm_maxs(
const bmtensor<Dtype>& tensorA,
const Dtype scalar,
bmtensor<Dtype>& tensor_result)
```

参数说明：tensorA，tensor_result 均是 bmtensor 类型，scalar 是 float32 类型。

2. Python API:

```
bm_maxs_float(tensorA,  scalar, tensor_result)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型，scalar 是标量类型。

3.10 bm_mins（TODO）

tensor_result = min(tensorA, scalar)，实现 tensorA 中每个元素都与 scalar 比较大小，取小值输出，支持数据类型：float32。

1. C++ API:

```
bm_mins(
const bmtensor<Dtype>& tensorA,
const Dtype scalar,
bmtensor<Dtype>& tensor_result)
```

参数说明：tensorA，tensor_result 均是 bmtensor 类型，scalar 是 float32 类型。

2. Python API:

```
bm_mins_float(tensorA, scalar, tensor_result)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型，scalar 是标量类型。

3.11 bm_axpy

tensor_result = scalar * tensorA + tensor_result，实现 tensorA 中每个元素都与 scalar 相乘，再与 tensor_result 每个元素相加，支持数据类型：float32。

1. C++ API:

```
bm_axpy(
const bmtensor<Dtype>& tensorA,
const Dtype scalar,
bmtensor<Dtype>& tensor_result)
```

参数说明：tensorA，tensor_result 均是 bmtensor 类型，scalar 是 float32 类型。

2. Python API:

```
bm_axpy_float(tensorA, scalar, tensor_result)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型，scalar 是标量类型。

3.12 bm_strideadd（TODO）

tensor_result = tensorA_i + tensorB_i，依据各个 dimation 上设置的 stride 值执行两个 tensor 相应 index 的元素加，支持数据类型：float32。

1. C++ API:

```
bm_strideadd(
const bmtensor<Dtype>& tensorA,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& tensor_result,
vector<int> stride)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型；stride 是 vector<int> 类型，用于指定 shape 每个维度上的 stride。

2. Python API:

```
bm_strideadd_float(tensorA,  tensorB, tensor_result, stride)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型；stride 是 intVector 类型，用于指定 shape 每个维度上的 stride。

3.13 bm_stridesub（TODO）

tensor_result = tensorA_i - tensorB_i，依据各个 dimation 上设置的 stride 值执行两个 tensor 相应 index 的元素减，支持数据类型：float32。

1. C++ API:

```
bm_stridesub(
const bmtensor<Dtype>& tensorA,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& tensor_result,
vector<int> stride)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型；stride 是 vector<int> 类型，用于指定 shape 每个维度上的 stride。

2. Python API:

```
bm_stridesub_float(tensorA,  tensorB, tensor_result, stride)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型；stride 是 intVector 类型，用于指定 shape 每个维度上的 stride。

3.14 bm_stridemul（TODO）

tensor_result = tensorA_i * tensorB_i，依据各个 dimation 上设置的 stride 值执行两个 tensor 相应 index 的元素乘，支持数据类型：float32。

1. C++ API:

```
bm_stridemul(
const bmtensor<Dtype>& tensorA,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& tensor_result,
vector<int> stride)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型；stride 是 vector<int> 类型，用于指定 shape 每个维度上的 stride。

2. Python API:

```
bm_stridemul_float(tensorA,  tensorB, tensor_result, stride)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型；stride 是 intVector 类型，用于指定 shape 每个维度上的 stride。

3.15 bm_stridemax（TODO）

tensor_result = max(tensorA_i, tensorB_i)，依据各个 dimation 上设置的 stride 值执行两个 tensor 相应 index 的元素比较，取大值输出，支持数据类型：float32。

1. C++ API:

```
bm_stridemax(
const bmtensor<Dtype>& tensorA,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& tensor_result,
vector<int> stride)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型；stride 是 vector<int> 类型，用于指定 shape 每个维度上的 stride。

2. Python API:

```
bm_stridemax_float(tensorA,  tensorB, tensor_result, stride)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型；stride 是 intVector 类型，用于指定 shape 每个维度上的 stride。

3.16 bm_strideadd_offset（TODO）

tensor_result = tensorA_i_offset + tensorB_i_offset，以各个 dimation 上设置的 offset 为起点，并依据设置的 stride 值执行两个 tensor 相应 index 的元素加，支持数据类型：float32。

1. C++ API:

```
bm_strideadd_offset(
const bmtensor<Dtype>& tensorA,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& tensor_result,
vector<int> stride,
vector<int> offset)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型；stride 是 vector<int> 类型，用于指定 shape 每个维度上的 stride；offset 是 vector<int> 类型，用于指定 shape 每个维度上的 offset。

2. Python API:

```
bm_strideadd_offset_float(tensorA,  tensorB, tensor_result, stride, offset)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型；stride 是 intVector 类型，用于指定 shape 每个维度上的 stride；offset 是 intVector 类型，用于指定 shape 每个维度上的 offset。

3.17 bm_stridesub_offset（TODO）

tensor_result = tensorA_i_offset - tensorB_i_offset，以各个 dimation 上设置的 offset 为起点，并依据设置的 stride 值执行两个 tensor 相应 index 的元素减，支持数据类型：float32。

1. C++ API:

```
bm_stridesub_offset(
const bmtensor<Dtype>& tensorA,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& tensor_result,
vector<int> stride,
vector<int> offset)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型；stride 是 vector<int> 类型，用于指定 shape 每个维度上的 stride；offset 是 vector<int> 类型，用于指定 shape 每个维度上的 offset。

2. Python API:

```
bm_stridesub_offset_float(tensorA,  tensorB, tensor_result, stride, offset)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型；stride 是 intVector 类型，用于指定 shape 每个维度上的 stride；offset 是 intVector 类型，用于指定 shape 每个维度上的 offset。

3.18 bm_stridemul_offset（TODO）

tensor_result = tensorA_i_offset * tensorB_i_offset，以各个 dimation 上设置的 offset 为起点，并依据设置的 stride 值执行两个 tensor 相应 index 的元素乘，支持数据类型：float32。

1. C++ API:

```
bm_stridemul_offset(
const bmtensor<Dtype>& tensorA,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& tensor_result,
vector<int> stride,
vector<int> offset)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型；stride 是 vector<int> 类型，用于指定 shape 每个维度上的 stride；offset 是 vector<int> 类型，用于指定 shape 每个维度上的 offset。

2. Python API:

```
bm_stridemul_offset_float(tensorA,  tensorB, tensor_result, stride, offset)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型；stride 是 intVector 类型，用于指定 shape 每个维度上的 stride；offset 是 intVector 类型，用于指定 shape 每个维度上的 offset。

3.19 bm_stridemax_offset（TODO）

tensor_result = max(tensorA_i_offset, tensorB_i_offset)，以各个 dimation 上设置的 offset 为起点，并依据设置的 stride 值执行两个 tensor 相应 index 的元素比较，取大值输出，支持数据类型：float32。

1. C++ API:

```
bm_stridemax_offset(
const bmtensor<Dtype>& tensorA,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& tensor_result,
vector<int> stride,
vector<int> offset)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型；stride 是 vector<int> 类型，用于指定 shape 每个维度上的 stride；offset 是 vector<int> 类型，用于指定 shape 每个维度上的 offset。

2. Python API:

```
bm_stridemax_offset_float(tensorA,  tensorB, tensor_result, stride, offset)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型；stride 是 intVector 类型，用于指定 shape 每个维度上的 stride；offset 是 intVector 类型，用于指定 shape 每个维度上的 offset。

3.20 bm_active

对输入的 tensor 使用激活函数进行处理，激活类型在 bmlang_common.h 文件中 ACTIVE_TYPE_e 定义，支持数据类型：float32。支持的 active type 如下：

```
typedef enum active_type {
  ACTIVE_TANH   ,
  ACTIVE_SIGMOID,
  ACTIVE_RELU   ,
  ACTIVE_EXP    ,
  ACTIVE_ELU    ,
  ACTIVE_SQRT   ,
  ACTIVE_SQUARE ,
  ACTIVE_RSQRT  ,
  ACTIVE_ABSVAL ,
  ACTIVE_LN, // (ln 对数)
  UNKNOWN_ACTIVE_TYPE
} ACTIVE_TYPE_e;
```

1. C++ API:

```
bm_active(
const bmtensor<Dtype>& tensorA,
bmtensor<Dtype>& tensor_result,
ACTIVE_TYPE_e active_type)
```

参数说明：tensorA，tensor_result 均是 bmtensor 类型；active_type 是 ACTIVE_TYPE_e 类型，指定激活函数的类型。

2. Python API:

```
bm_active_float(tensorA, tensor_result, active_type)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型；active_type 是 ACTIVE_TYPE_e 类型，指定激活函数的类型。

3.21 bm_arg

对输入 tensor 的指定的 axis 求最大或最小值，输出对应的 index，并将该 axis 的 dim 设置为 1，支持数据类型：float32。Arg 参数结构体如下：

```
typedef struct arg_op{
  int axis;
  int method;  //0->max, 1->min
} ARG_OP_t;
```

1. C++ API:

```
bm_arg(
const bmtensor<Dtype>& tensorA,
bmtensor<Dtype>& tensor_result,
ARG_OP_t arg_op)
```

参数说明：tensorA，tensor_result 均是 bmtensor 类型；arg_op 是 ARG_OP_t 类型，指定处理的轴和操作（max or min）。

2. Python API:

```
bm_arg_float(tensorA, tensor_result, arg_op)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型；arg_op 是 ARG_OP_t 类型，指定处理的轴和操作（max or min）。

3.22 bm_pool

对输入 tensor 进行池化处理，支持的数据类型：float32。池化参数如下：

```
typedef enum pooling_type {
  AVE,
  MAX,
  STOCHASTIC,
  UNKNOWN,
}  POOLING_TYPE_e;
typedef struct pooling_param {
  int stride_h;
  int stride_w;
  int pad_h = 0;
  int pad_w = 0;
  int kernel_h; // if global pooling, no need to specify kernel height and weight.
  int kernel_w;
  POOLING_TYPE_e pooling_type = AVE;
  bool is_global_pooling = false;
  bool is_ceil_mode = false;
}  POOLING_PARAM_t;
```

1. C++ API:

```
bm_pool(
const bmtensor<Dtype>& tensorA,
bmtensor<Dtype>& tensor_result,
const PoolingParam& pooling_param)
```

参数说明：tensorA，tensor_result 均是 bmtensor 类型；pooling_param 是 PoolingParam 类型，指定池化相关参数。

2. Python API:

```
bm_pool_float(tensorA, tensor_result, pooling_param)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型；pooling_param 是 PoolingParam 类型，指定池化相关参数。

3.23 bm_reduce

依据 reduce way，对输入的 tensor 做 reduce 操作，支持数据类型：float32。reduce 的算子类型如下：

```
typedef enum reduce_method {
  REDUCE_MEAN = 0,
  REDUCE_SUM = 1,
  REDUCE_MAX = 2,
  REDUCE_MIN = 3,
  REDUCE_PROD= 4
} REDUCE_METHOD_e;
//对被 reduce 的所有轴求平均，求和，求最大值，求最小值，求内积。
```

(continues on next page)

(continued from previous page)

```
//reduce 参数如下:
typedef struct reduce_op{
  REDUCE_METHOD_e reduce_method;
  int reduce_axis;
  int reduce_way;
} REDUCE_OP_t;
/*
reduce_method 指定 reduce 算子类型;
reduce_axis 指定 reduce 轴;
reduce_way 指定 reduce 策略 ——— 0: 对 reduce_axis 轴及其之后的轴进行 reduce 操作
                            1: 只对 reduce_axis 轴进行 reduce 操作
                            2: 对除 axis==1 (即 C 轴) 外的所有轴进行 reduce。
*/

//For example:
tensorA_shape = { 10, 20,30,40}, reduce_axis = 2
/*educe_way = 0 ->*/ tensor_result_shape = { 10, 20, 1, 1}
/*reduce_way = 1 ->*/ tensor_result_shape = { 10, 20, 1, 40}
/*reduce_way = 2 ->*/ tensor_result_shape = { 1, 20, 1, 1}
void bm_reduce(const bmtensor<Dtype>& tensorA, bmtensor<Dtype>& tensor_result,);
```

1. C++ API:

```
bm_reduce(
const bmtensor<Dtype>& tensorA,
bmtensor<Dtype>& tensor_result,
REDUCE_OP_t reduce_op)
```

参数说明：tensorA，tensor_result 均是 bmtensor 类型；reduce_op 是 REDUCE_OP_t 类型，指定 reduce 相关参数。

2. Python API:

```
bm_reduce_float(tensorA, tensor_result, reduce_op)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型；reduce_op 是 REDUCE_OP_t 类型，指定 reduce 相关参数。

2.5.5 4 Relation Op

4.1 bm_gt

tensor_result = tensorA > tensorB ? 1 : 0，两个 tensor 按元素比较，如果 A 大于 B 输出 1，否则输出 0，支持数据类型：float32。

1. C++ API:

```
bm_gt(
const bmtensor<Dtype>& tensorA ,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& result_tensor)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型。

2. Python API:

```
bm_gt_float(tensorA, tensorB, tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型。

4.2 bm_ge

tensor_result = tensorA >= tensorB ? 1 : 0，两个 tensor 按元素比较，如果 A 大于等于 B 输出 1，否则输出 0，支持数据类型：float32。

1. C++ API:

```
bm_ge(
const bmtensor<Dtype>& tensorA ,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& result_tensor)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型。

2. Python API:

```
bm_ge_float(tensorA, tensorB, tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型。

4.3 bm_lt

tensor_result = tensorA < tensorB ? 1 : 0，两个 tensor 按元素比较，如果 A 小于 B 输出 1，否则输出 0，支持数据类型：float32。

1. C++ API:

```
bm_lt(
const bmtensor<Dtype>& tensorA ,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& result_tensor)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型。

2. Python API:

```
bm_lt_float(tensorA, tensorB, tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型。

4.4 bm_le

tensor_result = tensorA <= tensorB ? 1 : 0，两个 tensor 按元素比较，如果 A 小于等于 B 输出 1，否则输出 0，支持数据类型：float32。

1. C++ API:

```
bm_le(
const bmtensor<Dtype>& tensorA ,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& result_tensor)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型。

2. Python API:

```
bm_le_float(tensorA, tensorB, tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型。

4.5 bm_eq

tensor_result = tensorA == tensorB ? 1 : 0，两个 tensor 按元素比较，如果 A 等于 B 输出 1，否则输出 0，支持数据类型：float32。

1. C++ API:

```
bm_eq(
const bmtensor<Dtype>& tensorA ,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& result_tensor)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型。

2. Python API:

```
bm_le_float(tensorA, tensorB, tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型。

4.6 bm_ne

tensor_result = tensorA != tensorB ? 1 : 0，两个 tensor 按元素比较，如果 A 不等于 B 输出 1，否则输出 0，支持数据类型：float32。

1. C++ API:

```
bm_ne(
const bmtensor<Dtype>& tensorA ,
const bmtensor<Dtype>& tensorB,
bmtensor<Dtype>& result_tensor)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor 类型。

2. Python API:

```
bm_ne_float(tensorA, tensorB, tensor_result)
```

参数说明：tensorA，tensorB，tensor_result 均是 bmtensor_float 类型。

2.5.6 5 Selection Op

5.1 bm_condselect

tensor_result = cond > 0 ? tensorB : tensorC, tensor cond 按元素与 0 比较，大于 0 时输出 tensorB 对应位置元素，否则输出 tensorC，支持数据类型：float32。cond 不能为常数，tensorC 和 tensorB 可以为常系数，需要设置 tensor type 为 COEFF_TENSOR 。

1. C++ API:

```
bm_condselect(
const bmtensor<Dtype>& cond,
const bmtensor<Dtype>& tensorB,
const bmtensor<Dtype>& tensorC,
bmtensor<Dtype>& tensor_result)
```

参数说明：cond，tensorB，tensorC，tensor_result 均是 bmtensor 类型。

2. Python API:

```
bm_condselect_float(cond, tensorB, tensorC, tensor_result)
```

参数说明：cond，tensorB，tensorC，tensor_result 均是 bmtensor_float 类型。

5.2 bm_select

select 有两种 method：

```
typedef enum select_op {
  GREATER_THAN,
  GREATER_EQUAL,
  UNKNOWN_SELECT_OP
} SELECT_OP_e;

select_opt == GREATER_THAN:
tensor_result = tensorA > tensor B  ?  tensorC : tensorD; //tensor 按元素比较，A 大于
B 时输出 tensorC 对应位置的元素值，否则输出 tensorD。

select_opt == GREATER_ EQUAL:
tensor_result = tensorA >= tensor B  ?  tensorC : tensorD; //tensor 按元素比较，A 大于
等于 B 时输出 tensorC 对应位置的元素值，否则输出 tensorD。
```

tensorA 和 tensorB 其中之一可以为常系数，需要设置 tensor type 为 COEFF_TENSOR。tensorC 和 tensorD 可以为常系数，需要设置 tensor type 为 COEFF_TENSOR。支持数据类型为：float32。

1. C++ API:

```
bm_select(
const bmtensor<Dtype>& tensorA,
const bmtensor<Dtype>& tensorB,
const bmtensor<Dtype>& tensorC,
const bmtensor<Dtype>& tensorD,
bmtensor<Dtype>& tensor_result,
SELECT_OP_e select_op)
```

参数说明：tensorA，tensorB，tensorC，tensorD，tensor_result 均是 bmtensor 类型；select_op 是 SELECT_OP_e 类型，指定 select method。

2. Python API:

```
bm_select_float(tensorA, tensorB, tensorC, tensorD, tensor_result, select_op)
```

参数说明：tensorA，tensorB，tensorC，tensorD，tensor_result 均是 bmtensor_float 类型; select_op 是 SELECT_OP_e 类型，指定 select method。

2.5.7 6 Data Manipulate Op

6.1 bm_permute

置换 tensor shape 的 axis，并相应进行数据的交换，支持数据类型：float32。例如：输入 shape 为（6，7，8，9），置换参数 order 为（1，3，2，0），则输出的 shape 为（7，9，8，6）。

1. C++ API:

```
bm_permute(
const bmtensor<Dtype>& tensorA,
bmtensor<Dtype>& tensor_result,
vector<int>& permute_order)
```

参数说明：tensorA，tensor_result 均是 bmtensor 类型；permute_order 是 vector<int> 类型，指定 permute order 参数。

2. Python API:

```
bm_permute_float(tensorA, tensor_result, permute_order)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型; permute_order 是 intVector 类型，指定 permute order 参数。

6.2 bm_extract

对 tensor 4 个维度上的数据做带 stride 的抽取操作，支持数据类型：float32。extract 参数如下：

```
typedef struct extract_op{
  int start[4];
  int stride[4];
  int end[4];
} EXTRACT_OP_t;
```

输出 shape 的每个 axis 计算方式为：(end[i] – start[i]) / stride[i]。例如某个 axis 的值为 10，start 为 1，end 为 7，stride 为 2，则 extract 之后输出 axis 的值为（7-1）/ 2 = 3，index 分别为 1，3，5。

1. C++ API:

```
bm_extract(
const bmtensor<Dtype>& tensorA,
bmtensor<Dtype>& tensor_result,
EXTRACT_OP_t extract_op)
```

参数说明：tensorA，tensor_result 均是 bmtensor 类型；extract_op 是 EXTRACT_OP_t 类型，指定 data extract method 参数。

2. Python API:

```
bm_extract_float(tensorA, tensor_result, extract_op)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型; extract_op 是 EXTRACT_OP_t 类型，指定 data extract method 参数。

6.3 bm_setdata

tensor_result = tensorA ，tensorA 的 tensor type 可以是 DATA_TENSOR 或 COEFF_TENSOR，支持广播，被广播的那一维度的维度值需要设置为 1，需要指定 tensor_result 的 shape，因为不能由输入推断出输出的维度，支持数据类型：float32。**Note:** 该 op 并不是 TPU 的实际操作。若 tensor type 是 DATA_TENSOR，则该操作只在 COMPILE/COMPUTE 模式下设置数据值，实际 TPU runtime 时不会是这里所设置的数据。若 tensor type 是 COEFF_TENSOR，则此时设置的数据是固定的，并且在 TPU runtime 时，也将是这里设置的数据。

1. C++ API:

```
bm_setdata(
const bmtensor<Dtype>& tensorA,
bmtensor<Dtype>& tensor_result)
```

参数说明：tensorA，tensor_result 均是 bmtensor 类型。

2. Python API:

```
bm_setdata_float(tensorA, tensor_result)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型。

6.4 bm_broadcast

将输入 tensor 数据广播到输出，支持数据类型：float32。broadcast 参数为：

```
typedef struct broadcast_op{
  int times[4];
} BROADCAST_OP_t;
//times[4] 分别表示每一维度被广播的份数。
```

1. C++ API:

```
bm_broadcast (
const bmtensor<Dtype>& tensorA,
bmtensor<Dtype>& tensor_result,
BROADCAST_OP_t broadcast_op)
```

参数说明：tensorA，tensor_result 均是 bmtensor 类型；broadcast_op 是 BROADCAST_OP_t 类型，指定每个维度广播的份数。

2. Python API:

```
bm_broadcast_float(tensorA, tensor_result, broadcast_op)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型；broadcast_op 是 BROADCAST_OP_t 类型，指定每个维度广播的份数。

6.5 bm_concat

对多个 tensor 在指定的轴上进行拼接，支持的数据类型：float32。

1. C++ API:

```
bm_concat (
const vector<bmtensor<Dtype>* >& tensorA,
bmtensor<Dtype>& tensor_result,
int axis_)
```

参数说明：tensorA 是 vector<bmtensor*> 类型，多个 tensor 的指针；tensor_result 是 bmtensor 类型；axis_ 是 int 类型，指定在哪一个轴上进行拼接。

2. Python API:

```
bm_concat_float(tfp_tensorA, tensor_result, axis_)
```

参数说明：tfp_tensorA 是 bmtfpVector 类型，多个 tensor 的指针；tensor_result 是 bmtensor_float 类型；axis_ 是标量，指定在哪一个轴上进行拼接。

6.6 bm_split

bm_concat 的逆操作。Split 参数：

```
typedef struct split_op{
  int size[MAX_SPLITNUM];
  int axis_;
  int num;
} SPLIT_OP_t;
//axis_: 指定在哪个轴上 split;
//num: 分裂份数;
//size[]: 每一份的大小（支持非平均分裂）。
```

1. C++ API:

```
bm_split (
const bmtensor<Dtype>& tensorA,
vector<bmtensor<Dtype>* >& tensor_result,
SPLIT_OP_t split_op)
```

参数说明：tensorA 是 bmtensor 类型；tensor_result 是 vector<bmtensor*> 类型，多个 tensor 的指针；split_op 是 SPLIT_OP_t 类型，指定 split 参数。

2. Python API:

```
bm_split_float(tensorA, tfp_vector_result, split_op)
```

参数说明：tensorA 是 bmtensor_float 类型；tfp_vector_result 是 bmtfpVector 类型，多个 tensor 的指针；split_op 是 SPLIT_OP_t 类型，指定 split 参数。

6.7 bm_reshape

对输入 tensor 做 reshape 的操作，支持数据类型：float32。void bm_reshape(const bmtensor<Dtype>& tensorA,bmtensor<Dtype>& tensor_result, RESHAPE_OP_t reshape_op); reshap 参数：

```
typedef struct reshape_op{
  int param[4];
} RESHAPE_OP_t;
/*
RESHAPE_OP_t  specify the output dimensions.
If some of the dimensions are set to 0,
the corresponding dimension from the bottom layer is used (unchanged).
Exactly one dimension may be set to -1, in which case its value is inferred from the␣
↪count of the bottom blob and the remaining dimensions.
*/
```

For example: tensorA_shape = (10, 20, 30, 40) param = (0, 10, -1, 10) tensor_result_shape = (10, 10, 240, 10)

1. C++ API:

```
bm_reshape (
const bmtensor<Dtype>& tensorA,
bmtensor<Dtype>& tensor_result,
RESHAPE_OP_t reshape_op)
```

参数说明：tensorA 和 tensor_result 均是 bmtensor 类型；reshape_op 是 RESHAPE_OP_t 类型，指定 reshape 参数。

2. Python API:

```
bm_reshape_float(tensorA, tensor_result, reshape_op)
```

参数说明：tensorA 和 tensor_result 均是 bmtensor_float 类型；reshape_op 是 RESHAPE_OP_t 类型，指定 reshape 参数。

6.8 bm_stridecpy（TODO）

tensor_result = tensorA_i，依据各个 dimation 上设置的 stride 值复制输入 tensor 相应 index 的元素到输出，支持数据类型：float32。

1. C++ API:

```
bm_stridecpy(
const bmtensor<Dtype>& tensorA,
bmtensor<Dtype>& tensor_result,
vector<int> stride)
```

参数说明：tensor，tensor_result 均是 bmtensor 类型；stride 是 vector<int> 类型，用于指定 shape 每个维度上的 stride。

2. Python API:

```
bm_stridecpy_float(tensorA,  tensor_result, stride)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型；stride 是 intVector 类型，用于指定 shape 每个维度上的 stride。

6.9 bm_stridecpy_offset（TODO）

tensor_result = tensorA_i_offset，以各个 dimation 上设置的 offset 为起点，并依据设置的 stride 值复制输入 tensor 相应 index 的元素到输出，支持数据类型：float32。

1. C++ API:

```
bm_stridecpy_offset(
const bmtensor<Dtype>& tensorA,
bmtensor<Dtype>& tensor_result,
vector<int> stride,
vector<int> offset)
```

参数说明：tensorA，tensor_result 均是 bmtensor 类型；stride 是 vector<int> 类型，用于指定 shape 每个维度上的 stride；offset 是 vector<int> 类型，用于指定 shape 每个维度上的 offset。

2. Python API:

```
bm_stridecpy_offset_float(tensorA, tensor_result, stride, offset)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型；stride 是 intVector 类型，用于指定 shape 每个维度上的 stride；offset 是 intVector 类型，用于指定 shape 每个维度上的 offset。

6.10 bm_lshift（TODO）

tensor_result = lshift(tensorA, shift_num)，对指定 axis 左移数据操作，支持数据类型：float32。举例：一个 tensor 是 n c h w 四个维度，当做 c 维度的逻辑左移时，高序号的 c 维度数据转到低序号的低维度数据，最高序号的 c 维度数据补 0。逻辑右移时则相反。比如对一个 tensor 做 c 维度上的逻辑左移 2 位，c 的范围为 0~c-1, 则 c=0, c=1 的数据丢弃，原来 Ci 上的数据等于移到了 Ci-2 位置上面。c-1 和 c-2 上的数据补 0。

1. C++ API:

```
bm_lshift(
const bmtensor<Dtype>& tensorA,
bmtensor<Dtype>& tensor_result,
int shift_dim,
int shift_num)
```

参数说明：tensorA，tensor_result 均是 bmtensor 类型；shift_dim 是 int 类型，用于指定左移的轴；shift_num 是 int 类型，用于指定左移位数。

2. Python API:

```
bm_lshift_float(tensorA, tensor_result, shift_dim, shift_num)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型；shift_dim 是标量类型，用于指定左移的轴；shift_num 是标量类型，用于指定左移位数。

6.11 bm_rshift（TODO）

tensor_result = rshift(tensorA, shift_num)，对指定 axis 右移数据操作，支持数据类型：float32。

1. C++ API:

```
bm_rshift(
const bmtensor<Dtype>& tensorA,
bmtensor<Dtype>& tensor_result,
int shift_dim,
int shift_num)
```

参数说明：tensorA，tensor_result 均是 bmtensor 类型；shift_dim 是 int 类型，用于指定右移的轴；shift_num 是 int 类型，用于指定右移位数。

2. Python API:

```
bm_rshift_float(tensorA, tensor_result, shift_dim, shift_num)
```

参数说明：tensorA，tensor_result 均是 bmtensor_float 类型；shift_dim 是标量类型，用于指定右移的轴；shift_num 是标量类型，用于指定右移位数。

2.5.8 7 Matrix Op

7.1 bm_matrixmul

tensor_result = mat_tensorA * mat_tensorB，对输入两个矩阵做乘法，例如：matA(m,k) * matB(k,n) 输出的 matrix 是 matR(m,n)，支持数据类型：float32。

1. C++ API:

```
bm_matrixmul(
const bmtensor<Dtype>& tensorA,
const bmtensor<Dtype>& tensorB,
const bmtensor<Dtype>& tensorC,
bmtensor<Dtype>& tensor_result)
```

参数说明：tensorA，tensorB ，tensorC，tensor_result 均是 bmtensor 类型。

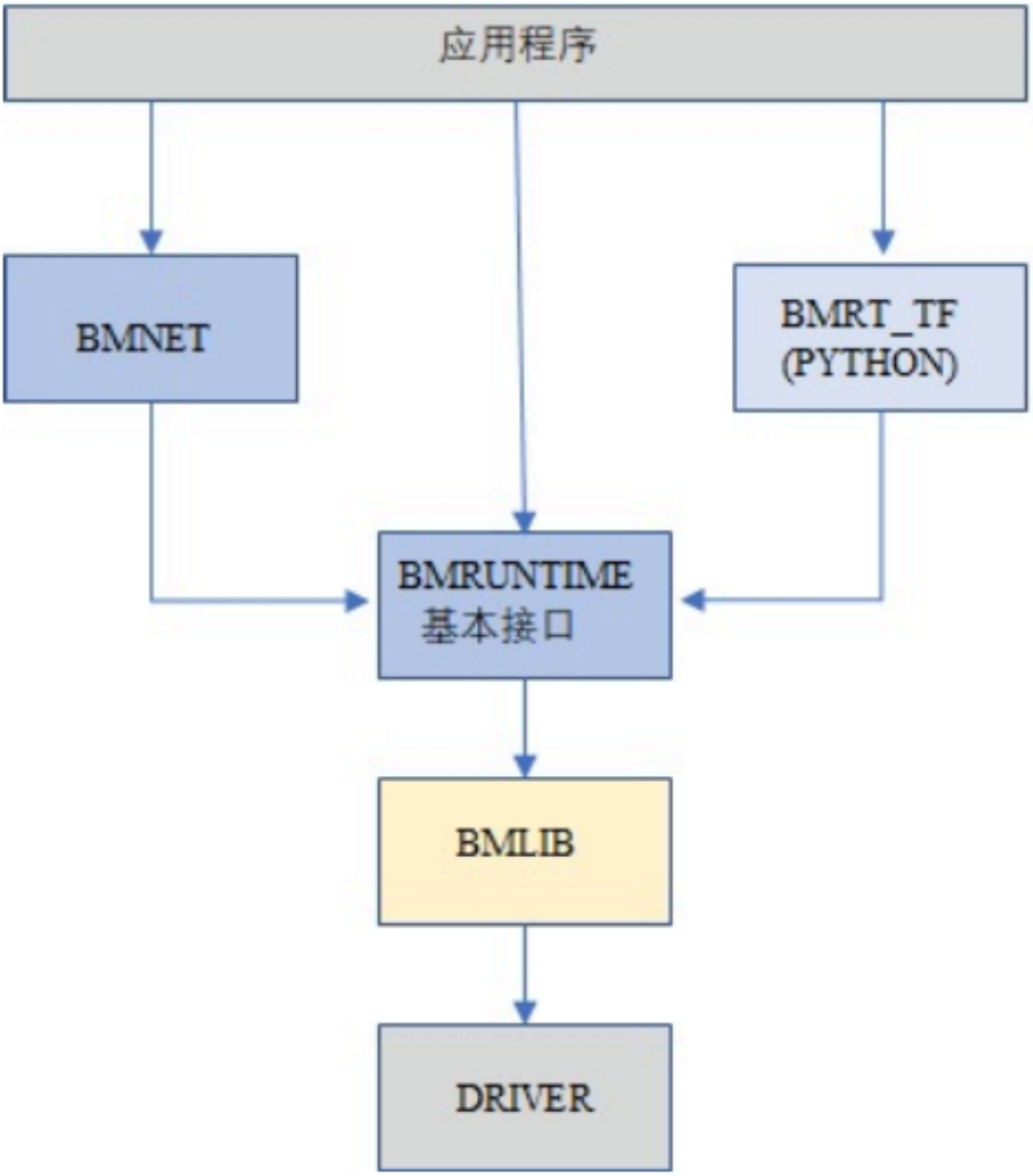
2. Python API:

```
bm_matrixmul_float(tensorA, tensorB , [tensorC,] tensor_result)
```

参数说明：tensorA，tensorB ，tensorC，tensor_result 均是 bmtensor_float 类型。特殊说明：tensorC 为可选，传参存在 tensorC，执行加 bias，否则不加，bias 个数等于右矩阵列数。

2.6 BMRuntime 使用

BMRUNTIME 用于读取 BMCompiler 的编译输出 (.bmodel)，驱动其在 BITMAIN TPU 芯片中执行。BMRUNTIME 向用户提供了丰富的接口，便于用户移植算法，其软件架构如下:



2.6.1 C Interface (Base)

基本的 BMRuntime 接口采用 C 语言，对应的头文件为 bmruntime_interface.h，对应的 lib 库为 libbmrt.so。

Create bmruntime

```
void* create_bmrt_helper(void *pbm_handle, const char *chip_type, int devid);
```

This API creates the bmruntime. It returns a void* pointer which is the pointer of bmruntime. This API can set device id.

Parameters

- **[in/out] pbm_handle** - BM runtime context. It is input if it had been created before. If it is NULL, it will be created as the output. This context is for using BMDNN and BMCV.
- **[in] chip_type** - Chipname, such as BM1682_PCIE, BM1682_SOC, BM1682_PCIE, BM1684_SOC.
- **[in] devid** - ID of device.

Returns

- **p_bmrt** - The pointer of bmruntime

```
#define create_bmruntime(pbm_handle, chip_type, ...) create_bmrt_helper(pbm_handle, chip_type,␣
↪(0, ##__VA_ARGS__))
```

This API creates the bmruntime. It returns a void* pointer which is the pointer of bmruntime. This API default uses the 0-th device.

Destroy bmruntime

```
void destroy_bmruntime(void* p_bmrt);
```

This API destroys the bmruntime.

Parametets

- **[in] p_bmrt** - Bmruntime that had been created.

Load bmodel

```
bool bmrt_load_bmodel(void* p_bmrt, const char *bmodel_path);
```

This API is to load bmodel created by BM compiler. After loading bmodel, we can run the inference of neuron network. Different with bmrt_load context, bmodel is a file. Bmodel is the pack of the context.

Parameters

- **[in] p_bmrt** - Bmruntime that had been created
- **[in] bmodel_path** - Bmodel file directory.

Returns

- **bool** - true: load bmodel success; false: load bmodel failed.

Show neuron network

```
void bmrt_show_neuron_network(void* p_bmrt);
```

To print the name of all neuron network

Parameters

- **[in] p_bmrt** - Bmruntime that had been created

Get network number

```
int bmrt_get_network_number(void* p_bmrt);
```

To get the number of neuron network in the bmruntime

Parameters

- [in] **p_bmrt** - Bmruntime that had been created

Returns

- **int** - The number of neuron networks.

Get network names

```
void bmrt_get_network_names(void* p_bmrt, const char*** network_names);
```

To get the names of all neuron network in the bmruntime

Parameters

- [in] **p_bmrt** - Bmruntime that had been created.
- [out] **network_names** - The names of all neuron networks. It should be declare as (const char** networks = NULL), and use as the param &networks_. After this API, user need to free(networks) if user do not need it.

Get network index

```
int bmrt_get_network_index(void* p_bmrt, const char* net_name);
```

To get the index of a network.

Parameters

- [in] **p_bmrt** - Bmruntime that had been created
- [in] **net_name** - The names of the neuron network.

Returns

- **int** - The index of the network

Get input tensors

```
void bmrt_get_input_tensor(void* p_bmrt, int net_idx, const char *net_name, int *input_num,␣
↪const char ***input_names);
```

This API can get the input number and input name of a neuron network. This API can be used after loading context or bmodel.

Parameters

- [in] **p_bmrt** - Bmruntime that had been created.
- [in] **net_idx** - Neuron network index of the context. This parameter is enable if net_name == NULL, otherwise it is disable.
- [in] **net_name** - The name of the neuron network. It can be NULL if we want to disable it.
- [out] **input_num** - The input number of the neuron network.
- [out] **input_names** - The input names of the neruon network. When we use this API, we can declare (const char** inputs = NULL), then use &inputs as this parameter. Bmruntime will create space for inputs, and we need to free(inputs) if we do not use it.

Get output tensors

```
void bmrt_get_output_tensor(void* p_bmrt, int net_idx, const char *net_name, int *output_num,␣
↪const char ***output_names);
```

This API can get the output number and output name of a neuron network. This API can only be used after loading context or bmodel.

Parameters

- [**in**] **p_bmrt** - Bmruntime that had been created
- [**in**] **net_idx** - Neuron network index of the context. This parameter is enable if net_name == NULL, otherwise it is disable.
- [**in**] **net_name** - The name of the neuron network. It can be NULL if we want to use net_idx.
- [**out**] **output_num** - The output number of the neuron network.
- [**out**] **output_names** - The output names of the neruon network. When we use this API, we can declare (const char** outputs = NULL), then use &outputs as this parameter. Bmruntime will create space for outputs, and we need to free(outputs) if we do not use it.

Get input shape

```
void bmrt_get_input_blob_max_nhw(void* p_bmrt, const char *tensor_name,
                                 int net_idx, const char *net_name,
                                 int * max_n, int * max_c, int * max_h, int * max_w);
```

To get the max shape of the tensor. This API must indicate the name of the input tensor and neuron network. When using the neuron network that is static-compiled once, max shape is the real shape of the neuron netork. When using the neuron network that is dynamic-compiled, max shape is max value we can set.

Parameters

- [**in**] **p_bmrt** - Bmruntime that had been created
- [**in**] **tensor_name** - The name of the tensor.
- [**in**] **net_idx** - Neuron network index of the context. This parameter is enable if net_name == NULL, otherwise it is disable.
- [**in**] **net_name** - The name of the neuron network. It can be NULL if we want to use net_idx.
- [**out**] **max_n** - The max batch number of the tensor.
- [**out**] **max_c** - The max channels of the tensor.
- [**out**] **max_h** - The max height of the tensor.
- [**out**] **max_w** - The max width of the tensor.

Get output shape

```
void bmrt_get_output_blob_max_nhw(void* p_bmrt, const char *tensor_name,
                                  int net_idx, const char *net_name,
                                  int * max_n, int * max_c, int * max_h, int * max_w);
```

To get the max shape of the tensor. This API must indicate the name of the output tensor and neuron network. When using the neuron network that is static-compiled once, max shape is the real shape of the neuron netork. When using the neuron network that is dynamic-compiled, max shape is max value we can set.

Parameters

- [**in**] **p_bmrt** - Bmruntime that had been created
- [**in**] **tensor_name** - The name of the tensor.
- [**in**] **net_idx** - Neuron network index of the context. This parameter is enable if net_name == NULL, otherwise it is disable.
- [**in**] **net_name** - The name of the neuron network. It can be NULL if we want to use net_idx.
- [**out**] **max_n** - The max batch number of the tensor.
- [**out**] **max_c** - The max channels of the tensor.
- [**out**] **max_h** - The max height of the tensor.
- [**out**] **max_w** - The max width of the tensor.

```
int bmrt_get_accurate_output_channel(void* p_bmrt, const char *tensor_name, int net_idx, const␣
↪char *net_name);
int bmrt_get_accurate_output_height(void* p_bmrt, const char *tensor_name, int net_idx, const␣
↪char *net_name);
int bmrt_get_accurate_output_width(void* p_bmrt, const char *tensor_name, int net_idx, const␣
↪char *net_name);
```

To get the output channel, height and width. **Note:** These API must be called after inference launch. It will block cpu program to wait for the finishing of network inference if the neuron network is dynamic-compiled.

Parameters

- [**in**] **p_bmrt** - Bmruntime that had been created.
- [**in**] **tensor_name** - The name of the tensor
- [**in**] **net_idx** - Neuron network index of the context. This parameter is enable if net_name == NULL, otherwise it is disable.
- [**in**] **net_name** - The name of the neuron network. It can be NULL if we want to use net_idx.

Returns

- **int** - The channels/height/width of the output tensor.

Get input data type

```
int bmrt_get_input_data_type(void* p_bmrt, const char *tensor_name, int net_idx, const char␣
↪*net_name);
```

To get the data type of the input tensor. The API is to get the data type of the tensor. Data type include float32, float16, int8, uint8, int16, uint16.

Parameters

- [**in**] **p_bmrt** - Bmruntime that had been created
- [**in**] **tensor_name** - The name of the tensor.
- [**in**] **net_idx** - Neuron network index of the context. This parameter is enable if net_name == NULL, otherwise it is disable.

- **[in] net_name** - The name of the neuron network. It can be NULL if we want to use net_idx.

Returns

- **int** - 0: float32, 1: float16, 2: int8, 3: uint8, 4: int16, 5: uint16.

Get output data type

```
int bmrt_get_output_data_type(void* p_bmrt, const char *tensor_name, int net_idx, const char␣
↪*net_name);
```

To get the data type of the output tensor. The API is to get the data type of the tensor. Data type include float32, float16, int8, uint8, int16, uint16.

Parameters

- **[in] p_bmrt** - Bmruntime that had been created
- **[in] tensor_name** - The name of the tensor.
- **[in] net_idx** - Neuron network index of the context. This parameter is enable if net_name == NULL, otherwise it is disable.
- **[in] net_name** - The name of the neuron network. It can be NULL if we want to use net_idx.

Returns

- **int** - 0: float32, 1: float16, 2: int8, 3: uint8, 4: int16, 5: uint16.

Get input store mode

```
int bmrt_get_input_gmem_stmode(void* p_bmrt, const char *tensor_name, int net_idx, const char␣
↪*net_name);
```

To get the data store mode of the input tensor. The API is to get the data store mode of the tensor. Data store mode includes 1N mode, 2N mode, 4N mode. 1N mode is the normal store mode we know.

Parameters: - [in] p_bmrt - Bmruntime that had been created.

- **[in] tensor_name** - The name of the tensor.
- **[in] net_idx** - Neuron network index of the context. This parameter is enable if net_name == NULL, otherwise it is disable.
- **[in] net_name** - The name of the neuron network. It can be NULL if we want to use net_idx.

Returns

- **int** - 0: 1N, 1: 2N, 2: 4N

Get output store mode

```
int bmrt_get_output_gmem_stmode(void* p_bmrt, const char *tensor_name, int net_idx, const char␣
↪*net_name);
```

To get the data store mode of the output tensor. The API is to get the data store mode of the tensor. Data store mode includes 1N mode, 2N mode, 4N mode. 1N mode is the normal store mode we know.

Parameters: - [in] p_bmrt - Bmruntime that had been created.

- [in] **tensor_name** - The name of the tensor.
- [in] **net_idx** - Neuron network index of the context. This parameter is enable if net_name == NULL, otherwise it is disable.
- [in] **net_name** - The name of the neuron network. It can be NULL if we want to use net_idx.

Returns

- **int** - 0: 1N, 1: 2N, 2: 4N

Input shape change

```
bool bmrt_can_batch_size_change(void* p_bmrt, int net_idx, const char *net_name);
```

To judge if the neuron network after compiling can change batch size.

Parameters

- [in] **p_bmrt** - Bmruntime that had been created.
- [in] **net_idx** - Neuron network index of the context. This parameter is enable if net_name == NULL, otherwise it is disable.
- [in] **net_name** - The name of the neuron network. It can be NULL if we want to use net_idx.

Returns

- **bool** - ture: The neuron network after compiling can change batch size. false: The neuron network after compiling can not change batch size.

```
bool bmrt_can_height_and_width_change(void* p_bmrt, int net_idx, const char *net_name);
```

To judge if the neuron network after compiling can change input height and width

Parameters

- [in] **p_bmrt** - Bmruntime that had been created.
- [in] **net_idx** - Neuron network index of the context. This parameter is enable if net_name == NULL, otherwise it is disable.
- [in] **net_name** - The name of the neuron network. It can be NULL if we want to use net_idx.

Returns

- **bool** - ture: The neuron network after compiling can change input height and width. false: The neuron network after compiling can not change input height and width.

Launch computation

```
bool bmrt_launch_nhw(void* p_bmrt, int net_idx, const char *net_name,
                     const void* input_tensors, int input_num,
                     const void* output_tensors, int output_num,
                     int n, int h, int w);
```

To launch the inference of the neuron network with setting input n, h, w. This API supports the neuron nework that is static-compiled or dynamic-compiled. After calling this API, inference on TPU is launched. And the CPU program will not be blocked. **Note:** This API only support one input

Parameters

- [in] **p_bmrt** - Bmruntime that had been created.

- [in] **net_idx** - Neuron network index of the context. This parameter is enable if net_name == NULL, otherwise it is disable.
- [in] **net_name** - The name of the neuron network. It can be NULL if we want to use net_idx.
- [in] **input_tensors** - The pointer of the input devce memory. It should be bm_device_mem_t*. input_tensors[i] must be created by malloc device memory.
- [in] **input_num** - The input number of the neuron network.
- [out] **output_tensors** - The pointer of the output devce memory. It should be bm_device_mem_t*. output_tensors[i] must be created by malloc device memory.
- [in] **output_num** - The output number of the neuron network.
- [in] **n** - The batch number of the neuron network.
- [in] **h** - The input height of the neuron network.
- [in] **w** - The input width of the neuron network.

Returns

- **bool** - true: Launch success. false: Launch failed.

```
bool bmrt_launch_shape(void* p_bmrt, int net_idx, const char *net_name,
                       const void* input_tensors, int input_num,
                       const int* input_dim, const int* input_shape,
                       const void* output_tensors, int output_num);
```

To launch the inference of the neuron network with setting input shape. This API supports the neuron nework that is static-compiled or dynamic-compiled. After calling this API, inference on TPU is launched. And the CPU program will not be blocked. This API support multiple inputs.

Parameters

- [in] **p_bmrt** - Bmruntime that had been created.
- [in] **net_idx** - Neuron network index of the context. This parameter is enable if net_name == NULL, otherwise it is disable.
- [in] **net_name** - The name of the neuron network. It can be NULL if we want to use net_idx.
- [in] **input_tensors** - The pointer of the input devce memory. It should be bm_device_mem_t*. input_tensors[i] must be created by malloc device memory.
- [in] **input_num** - The input number of the neuron network.
- [out] **output_tensors** - The pointer of the output devce memory. It should be bm_device_mem_t*. output_tensors[i] must be created by malloc device memory.
- [in] **output_num** - The output number of the neuron network.
- [in] **input_dim** - The dimension of each input.
- [in] **input_shape** - The shape of each input. If the dimension of the i-th input is x, input_shape[i] has x elements. For example, 4 dimension will has 4 elements, input_shape[i][0] is N, input_shape[i][1] is C, input_shape[i][2] is H, input_shape[i][3] is W.

Returns

- **bool** - true: Launch success. false: Launch failed.

```
bool bmrt_launch_cpu_data(void* p_bmrt, int net_idx, const char *net_name,
                      void* input_tensors, int input_num,
                      void* output_tensors, int output_num,
                      const int* in_shape, int* out_shape);
```

To launch the inference of the neuron network with cpu data. This API supports the neuron nework that is static-compiled or dynamic-compiled. After calling this API, inference on TPU is launched. And the CPU program will be blocked.

Parameters

- [**in**] **p_bmrt** - Bmruntime that had been created
- [**in**] **net_idxi** - Neuron network index of the context. This parameter is enable if net_name == NULL, otherwise it is disable.
- [**in**] **net_name** - The name of the neuron network. It can be NULL if we want to use net_idx.
- [**in**] **input_tensors** - The cpu pointer of the input.
- [**in**] **input_num** - The input number of the neuron network.
- [**out**] **output_tensors** - The cpu pointer of the output.
- [**in**] **output_num** - The output number of the neuron network.
- [**in**] **in_shape** - The input shape of the neuron network. Support multiple inputs. shape format (net0_n, net0_c, net0_h, net0_w, net1_n, net1_c, net1_h, net1_w)
- [**out**] **out_shape** - The output shape of the neuron network. Support multiple outputs. Bmruntime will set out_shape.

Returns

- **bool** - true: Launch success. false: Launch failed.

```
bool bmrt_launch_nhw_stmode(void* p_bmrt, int net_idx, const char *net_name,
                            const void* input_tensors, int input_num,
                            const void* output_tensors, int output_num,
                            int n, int h, int w,
                            int* in_stmode, int* out_stmode);
```

To launch the inference of the neuron network with setting input n/h/w and store mode. This API supports the neuron nework that is static-compiled or dynamic-compiled. After calling this API, inference on TPU is launched. And the CPU. program will not be blocked. **Note:** This API only support one input of the neuron network.

Parameters

- [**in**] **p_bmrt** - Bmruntime that had been created.
- [**in**] **net_idx** - Neuron network index of the context. This parameter is enable if net_name == NULL, otherwise it is disable.
- [**in**] **net_name** - The name of the neuron network. It can be NULL if we want to use net_idx.
- [**in**] **input_tensors** - The pointer of the input devce memory. It should be bm_device_mem_t*. input_tensors[i] must be created by malloc device memory.
- [**in**] **input_num** - The input number of the neuron network.
- [**out**] **output_tensors** - The pointer of the output devce memory. It should be bm_device_mem_t*. output_tensors[i] must be created by malloc device memory.
- [**in**] **output_num** - The output number of the neuron network.
- [**in**] **n** - The batch number of the neuron network.
- [**in**] **h** - The input height of the neuron network.
- [**in**] **w** - The input width of the neuron network.
- [**in**] **in_stmode** - User define the store mode of the input.

- [in] **out_stmode** - User define the store mode of the output.

Returns

- **bool** - true: Launch success. false: Launch failed.

```
bool bmrt_launch_shape_stmode(void* p_bmrt, int net_idx, const char *net_name,
                              const void* input_tensors, int input_num,
                              const int* input_dim, const int* input_shape,
                              const void* output_tensors, int output_num,
                              int* in_stmode, int* out_stmode);
```

To launch the inference of the neuron network with setting input shape and store mode. This API supports the neuron nework that is static-compiled or dynamic-compiled After calling this API, inference on TPU is launched. And the CPU program will not be blocked. This API support multiple inputs of the neuron network.

Parameters

- [in] **p_bmrt** - Bmruntime that had been created.
- [in] **net_idx** - Neuron network index of the context. This parameter is enable if net_name == NULL, otherwise it is disable.
- [in] **net_name** - The name of the neuron network. It can be NULL if we want to use net_idx.
- [in] **input_tensors** - The pointer of the input devce memory. It should be bm_device_mem_t*. input_tensors[i] must be created by malloc device memory.
- [in] **input_num** - The input number of the neuron network.
- [out] **output_tensors** - The pointer of the output devce memory. It should be bm_device_mem_t*. output_tensors[i] must be created by malloc device memory.
- [in] **output_num** - The output number of the neuron network.
- [in] **input_dim** - The dimension of each input.
- [in] **input_shape** - The shape of each input. If the dimension of the i-th input is x, input_shape[i] has x elements. For example, 4 dimension will has 4 elements, input_shape[i][0] is N, input_shape[i][1] is C, input_shape[i][2] is H, input_shape[i][3] is W.
- [in] **in_stmode** - User define the store mode of the input.
- [in] **out_stmode** - User define the store mode of the output.

Returns

- **bool** - true: Launch success. false: Launch failed.

Malloc device memory

```
int bmrt_malloc_device_byte(void* p_bmrt, bm_device_mem_t *pmem, unsigned int size);
```

To allocate device memory by byte size.

Parameters

- [in] **p_bmrt** - Bmruntime that had been created.
- [in] **pmem** - Device memory. If use declare device memory as (bm_device_mem_t mem), use it as parameter (&mem).
- [in] **size** - The number of byte.

Returns

- **int** - 0: Success. other: fail.

```
int bmrt_malloc_neuron_device(void* p_bmrt, bm_device_mem_t *pmem, int n, int c, int h, int w);
```

To allocate device memory by neuron. The size of the device memory by this API is (n * c * h * w * sizeof(float))

Parameters

- [**in**] **p__bmrt** - Bmruntime that had been created.
- [**in**] **pmem** - Device memory. If use declare device memory as (bm_device_mem_t mem), use it as parameter (&mem).
- [**in**] **n** - The batch size.
- [**in**] **c** - The channels.
- [**in**] **h** - The height.
- [**in**] **w** - The width.

Returns

- **int** - 0: Success. other: fail.

Free device memory

```
void bmrt_free_device(void* p_bmrt, bm_device_mem_t mem);
```

To free the device memory.

Parameters

- [**in**] **p__bmrt** - Bmruntime that had been created.
- [**in**] **pmem** - Device memory. If use declare device memory as (bm_device_mem_t mem), use it as parameter (&mem).

Data copy

```
int bmrt_memcpy_s2d(void* p_bmrt, bm_device_mem_t dst, void* src);
```

To copy data from system to device. The system memory is allocated by user. The data size transfered by this API is equal to the device memory capacity that user had allocated.

Parameters

- [**in**] **p__bmrt** - Bmruntime that had been created.
- [**out**] **dst** - destination device memory.
- [**in**] **src** - Source system memory.

Returns

- **int** - 0: Success. other: fail.

```
int bmrt_memcpy_d2s(void* p_bmrt, void *dst, bm_device_mem_t src);
```

To copy data from device to system. The system memory is allocated by user. The data size transfered by this API is equal to the device memory capacity that user had allocated.

Parameters

- [**in**] **p__bmrt** - Bmruntime that had been created.

- [**out**] **dst** - destination system memory.
- [**in**] **src** - Source device memory.

Returns

- **int** - 0: Success. other: fail.

```
int bmrt_memcpy_s2d_withsize(void* p_bmrt, bm_device_mem_t dst, void* src, unsigned int size);
```

To copy data from system to device with setting size. The system memory is allocated by user. The data size transfered by this API can be set by user.

Parameters

- [**in**] **p_bmrt** - Bmruntime that had been created.
- [**out**] **dst** - destination device memory.
- [**in**] **src** - Source system memory.
- [**in**] **size** - The byte size.

Returns

- **int** - 0: Success. other: fail.

```
int bmrt_memcpy_d2s_withsize(void* p_bmrt, void *dst, bm_device_mem_t src, unsigned int size);
```

To copy data from device to system with setting size. The system memory is allocated by user. The data size transfered by this API can be set by user.

Parameters

- [**in**] **p_bmrt** - Bmruntime that had been created.
- [**out**] **dst** - destination system memory.
- [**in**] **src** - Source device memory.
- [**in**] **size** - The byte size.

Returns

- **int** - 0: Success. other: fail.

```
int bmrt_memcpy_s2d_withsize_offset(void* p_bmrt, bm_device_mem_t dst, void* src, unsigned int␣
↪size, unsigned int dev_offset);
```

To copy data from system to device with setting size and device memory offset. The system memory is allocated by user. The data size transfered by this API can be set by user. And device memory offset can be also set by user. Data from device memory (address + offset) is transfered.

Parameters

- [**in**] **p_bmrt** - Bmruntime that had been created.
- [**out**] **dst** - destination device memory.
- [**in**] **src** - Source system memory.
- [**in**] **size** - The byte size.
- [**in**] **dev_offset** - Offset of device memory. Unit is byte.

Returns

- **int** - 0: Success. other: fail.

```
int bmrt_memcpy_d2s_withsize_offset(void* p_bmrt, void *dst, bm_device_mem_t src, unsigned int␣
↪size, unsigned int dev_offset);
```

To copy data from device to system with setting size and device memory offset. The system memory is allocated by user. The data size transfered by this API can be set by user. And device memory offset can be also set by user. Data from device memory (address + offset) is transfered.

Parameters

- [**in**] **p_bmrt** - Bmruntime that had been created.
- [**out**] **dst** - destination system memory.
- [**in**] **src** - Source device memory.
- [**in**] **size** - The byte size.
- [**in**] **dev_offset** - Offset of device memory. Unit is byte.

Returns

- **int** - 0: Success. other: fail.

Program synchronize

```
int bmrt_thread_sync(void* p_bmrt);
```

To synchronize cpu program to device. This API is called when user program want to wait for the finishing of device.

Parameters

- [**in**] **p_bmrt** - Bmruntime that had been created.

Returns

- **int** - 0: Success. other: fail.

2.6.2 C++ Interface (Caffe like)

类 Caffe 的 Inference C++ 接口。对应的头文件为 bmblob.h, bmcnnctx.h, bmnet.h，对应的 lib 库为 libbmrt.so。详细接口和注解请见 bmblob.h, bmcnnctx.h, bmnet.h。

2.7 bmodel 使用

bmodel 是面向比特大陆 TPU 处理器的深度神经网络模型文件格式。通过模型编译器工具 (如 bmnetc/bmnett 等) 生成，包含一个至多个网络的参数信息，如输入输出等信息。并在 runtime 阶段作为模型文件被加载和使用。通过 bm_model.bin 工具，可以查看 bmodel 文件的参数信息，可以将多个网络 bmodel 分解成多个单网络的 bmodel，也可以将多个网络的 bmodel 合并成一个 bmodel。

1. 使用方法

- 查看简要信息

  ```
  bm_model.bin --info xxx.bmodel
  ```

- 查看详细参数信息

  ```
  bm_model.bin --print xxx.bmodel
  ```

- 分解

  ```
  bm_model.bin --extract xxx.bmodel
  ```

  将一个包含多个网络的 bmodel 分解成只包含一个网络的各个 bmodel，分解出来的 bmodel 命名为 bm_net0.bmodel、bm_net1.bmodel ……

- 合并

  ```
  bm_model.bin --combine a.bmodel b.bmodel c.bmodel -o abc.bmodel
  ```

  将多个 bmodel 合并成一个 bmodel，-o 用于指定输出文件名，如果没有指定，则默认命名为 compilation.bmodel

CHAPTER

THREE

NNTOOLCHAIN 示例程序

3.1 示例代码

3.1.1 BMLang Examples

Example with BMLang Python

This example implements the standard LSTM algorithm independently. BMLang has two mode: COMPILE mode and DEBUG mode. DEBUG mode is for checking whether the BMLang program is correct or not. COMPILE mode is to generate bmodel. After COMPILE mode, we can use the BMRuntime to run this lstm bmodel in BMTPU device.

```
#!/usr/bin/env python

import numpy as np
from bmlang import *

class Lstm:
  def __init__(self, x_len, h_len):
    self.x_len = x_len
    self.h_len = h_len

    ## Vector for recording the reference data, it is of use when using BMLANG_BOTH
    ## We can check the result during compiling
    ## Here is optional
    # self.refTenVec = bmtfPtrVector()
    # self.inpTenVec = bmtfPtrVector()

    ## get the coefficient data
    self.Ui, self.Wi, self.Bi = self.init_coeff('lstm_coeff_i')
    self.Uf, self.Wf, self.Bf = self.init_coeff('lstm_coeff_f')
    self.Uo, self.Wo, self.Bo = self.init_coeff('lstm_coeff_o')
    self.Ug, self.Wg, self.Bg = self.init_coeff('lstm_coeff_g')

  def init_coeff(self, coeff_name):
    uSize = self.x_len * self.h_len
    wSize = self.h_len * self.h_len
    bSize = self.h_len
    ## Here we use random data. In fact, here should use real trained data
    u = np.random.uniform(-np.sqrt(1.0 / self.x_len), np.sqrt(1.0 / self.x_len), uSize)
    w = np.random.uniform(-np.sqrt(1.0 / self.h_len), np.sqrt(1.0 / self.h_len), wSize)
    b = np.random.uniform(-np.sqrt(1.0 / self.h_len), np.sqrt(1.0 / self.h_len), bSize)
    bmlang_print('u = ', u)
    bmlang_print('w = ', w)
    bmlang_print('b = ', b)
    uArray = floatArray(uSize)
    wArray = floatArray(wSize)
```

(continues on next page)

(continued from previous page)

```
    bArray = floatArray(bSize)
    for ui in range(0, uSize):
      uArray[ui] = float(u[ui])
    for wi in range(0, wSize):
      wArray[wi] = float(w[wi])
    for bi in range(0, bSize):
      bArray[bi] = float(b[bi])

    ## Create coefficient tensors, including uTensor, wTensor and bTensor
    ## A bmtensor_float params include the name and shapes.
    ## the shape dimension can be 8 at most
    uTensor = bmtensor_float(coeff_name + '_u', self.x_len, self.h_len)
    wTensor = bmtensor_float(coeff_name + '_w', self.h_len, self.h_len)
    bTensor = bmtensor_float(coeff_name + '_b', 1, self.h_len)
    ## declare these tensors are coefficient tensor
    ## This mean these data are fixed
    ## If not declare, this data must feed during runtime
    uTensor.set_tensor_type(COEFF_TENSOR)
    wTensor.set_tensor_type(COEFF_TENSOR)
    bTensor.set_tensor_type(COEFF_TENSOR)
    ## feed data to these tensors
    uTensor.fill_data(uArray)
    wTensor.fill_data(wArray)
    bTensor.fill_data(bArray)
    return uTensor, wTensor, bTensor

  ## The following is the LSTM computation decribed by BMLang Operators
  def forward(self, x, T):
    ## Create come intermediate bmtensors for computation
    ## We must set the name, but we can not always set shapes
    ## because the shapes of these intermediate tensors can be inferred
    iTensor = bmtensor_float('i_tensor')
    fTensor = bmtensor_float('f_tensor')
    oTensor = bmtensor_float('o_tensor')
    gTensor = bmtensor_float('g_tensor')
    tmpTensor = bmtensor_float('tmp_tensor')

    ## Create input x tensor, the shape must be set, because it is the input
    xTensor = bmtensor_float('x_tensor', 1, self.x_len * T)
    ## filling the data for input
    xTensor.set_data(x)
    ## create a vector to save multiple bm_tensors
    xTenVec = bmtfPtrVector()
    s_op_t = SPLIT_OP_t() ## OP to use bm_split_float
    s_size = intArray(T)  ## a int array
    xTenList = []
    for i in range(0, T):
      s_xTensor = bmtensor_float('x_tensor_' + str(i), 1, self.x_len)
      xTenList.append(s_xTensor)
      xTenVec.push_back(xTenList[i].this)
    s_size[0] = T
    s_op_t.size = s_size
    #s_op_t.num = T
    s_op_t.num = 1
    s_op_t.axis_ = 1

    bm_split_float(xTensor, xTenVec, s_op_t)

    ## Create other intermediate tensors
    inpTenList = []
    inpTenList.append(xTensor)
```

(continues on next page)

(continued from previous page)

```
    phTensor = bmtensor_float('h_tensor_t-1', 1, self.h_len)
    inpTenList.append(phTensor)
    pcTensor = bmtensor_float('c_tensor_t-1', 1, self.h_len)
    inpTenList.append(pcTensor)

    ## Recode the input data, it is of use when using BMLANG_BOTH
    ## We can check the result during compiling
    ## Here is optional
    # for j in range(len(inpTenList)):
    #  self.inpTenVec.push_back(inpTenList[j].this)

    hTenList = []
    for t in range(0, T):
      hTensor = bmtensor_float('h_tensor_t' + str(t), 1, self.h_len)
      cTensor = bmtensor_float('c_tensor_t' + str(t), 1, self.h_len)

      bm_matrixmul_float(xTenVec[t], self.Ui, self.Bi, iTensor)
      bm_matrixmul_float(phTensor, self.Wi, tmpTensor)
      bm_add_float(iTensor, tmpTensor, iTensor)
      bm_active_float(iTensor, iTensor, ACTIVE_SIGMOID)

      bmlang_print('i gate: ', iTensor.data_at(0, 0))

      bm_matrixmul_float(xTenVec[t], self.Uf, self.Bf, fTensor)
      bm_matrixmul_float(phTensor, self.Wf, tmpTensor)
      bm_add_float(fTensor, tmpTensor, fTensor)
      bm_active_float(fTensor, fTensor, ACTIVE_SIGMOID)

      bm_matrixmul_float(xTenVec[t], self.Uo, self.Bo, oTensor)
      bm_matrixmul_float(phTensor, self.Wo, tmpTensor)
      bm_add_float(oTensor, tmpTensor, oTensor)
      bm_active_float(oTensor, oTensor, ACTIVE_SIGMOID)

      bm_matrixmul_float(xTenVec[t], self.Ug, self.Bg, gTensor)
      bm_matrixmul_float(phTensor, self.Wg, tmpTensor)
      bm_add_float(gTensor, tmpTensor, gTensor)
      bm_active_float(gTensor, gTensor, ACTIVE_TANH)

      bmlang_print('g gate: ', gTensor.data_at(0, 0))

      bm_mul_float(pcTensor, fTensor, cTensor)
      bm_mul_float(gTensor, iTensor, tmpTensor)
      bm_add_float(cTensor, tmpTensor, cTensor)

      bmlang_print('c tensor: ', cTensor.data_at(0, 0))

      bm_active_float(cTensor, tmpTensor, ACTIVE_TANH)
      bm_mul_float(tmpTensor, oTensor, hTensor)

      bmlang_print('h tensor: ', hTensor.data_at(0, 0))

      phTensor = bmtensor_float(hTensor)
      pcTensor = bmtensor_float(cTensor)

      hTenList.append(hTensor)

    ## Recode the reference data, it is of use when using BMLANG_BOTH
    ## We can check the result during compiling
    ## Here is optional
    ## for i in range(len(hTenList)):
    ##   self.refTenVec.push_back(hTenList[i].this)
```

(continues on next page)

(continued from previous page)

```
    return hTenList

def compile(self):
  ## It is of use when using BMLANG_BOTH, it can check result when compiling
  ## When using this, we must turn on sefl.inpTenVec and self.refTenVec above
  # bmlang_compile_with_result_check('lstm', 2, True, self.inpTenVec, self.refTenVec)

  ## The funciton only compile, and generate bmodel
  bmlang_compile('lstm', 2, True)

def forward_cpu(self, x, T):
  ## Here we can call the cpu implementation for lstm
  ## This is only for debug whether lstm BMLang computation is corrent or wrong
  result = lstm(x, T)
  return result

PYBMLANG_DEBUG = False
def bmlang_print(*arg):
  if PYBMLANG_DEBUG:
    print('PYBMLANG LOG: ', *arg, end = "\n\n")
  else:
    pass

def debug_mode(lstm, x, T):
  set_mode(BMLANG_COMPILE) # This only compile
  ## set_mode(BMLANG_BOTH) # This can compile and compare the results
  oupList = lstm.forward(inpArray, T)
  ref = lstm.forward_cpu(inpArray, T)
  correct = compare_data(oupList, ref)
  if correct is True:
    print("LSTM with bmlang is right.")
  else :
    print("LSTM with bmlang compuation is wrong")

def compile_mode(lstm, x, T):
  set_mode(BMLANG_COMPUTE) # This is only for debug whether BMLANG lstm is right or not
  oupList = lstm.forward(inpArray, T)
  lstm.compile()

def lstm_main(xLen, hLen, T, mode):
  ## initialize the bmlang
  bmlang_init('BM1682', BMLANG_BOTH)
  inp = np.random.uniform(-10, 10, xLen * T)
  #inp = np.ones(xLen)
  bmlang_print(inp)
  inpArray = floatArray(xLen * T)
  for i in range(0, xLen * T):
    inpArray[i] = float(inp[i])
  lstm = Lstm(xLen, hLen)
  if mode == "compile":
    compile_mode(lstm, inpArray, T)
  else :
    debug_mode(lstm, inpArray, T)
  ## finish bmlang independently
  bmlang_deinit()

if __name__ == '__main__':
  lstm_main(1056, 896, 10, "compile")
  bmlang_print('LSTM IS FINISHED!')
```

3.1.2 BMRuntime examples

Example with basic C interface

该 example 读取网络的 reference 输入数据，启动 inference，最后将 TPU inference 结果与 reference 输出数据进行比对。

```
void bmrt_test()
{
  //create bmruntime
  void* p_bmrt = create_bmrt_helper(NULL, CHIPNAME.c_str(), dev_id);

  // load bmodel
  string bmodel_path = CONTEXT_DIR + "/compilation.bmodel";
  bool flag = bmrt_load_bmodel(p_bmrt, bmodel_path.c_str());
  if(!flag) {
    printf("Load bmodel[%s] failed\n", bmodel_path.c_str());
    exit(-1);
  }

  //get network number
  int net_num = bmrt_get_network_number(p_bmrt);

  char **net_names;
  bmrt_get_network_names(p_bmrt, (const char***)&net_names);
  string input_ref_dir = CONTEXT_DIR + "/" + INPUT_REF_DATA;
  string output_ref_dir = CONTEXT_DIR + "/" + OUTPUT_REF_DATA;

  FILE* f_input_ref = fopen(input_ref_dir.c_str(), "rb");
  if(f_input_ref == NULL) {
    printf("Error: cannot open file %s\n", input_ref_dir.c_str());
    exit(-1);
  }
  FILE* f_output_ref = fopen(output_ref_dir.c_str(), "rb");
  if(f_output_ref == NULL) {
    printf("Error: cannot open file %s\n", output_ref_dir.c_str());
    exit(-1);
  }

  const char** input_tensor_names = NULL;
  int input_tensor_num = 0;
  const char** output_tensor_names = NULL;
  int output_tensor_num = 0;
  for(int net_idx = 0; net_idx < net_num; ++net_idx) {
    printf("==> running network #%d: '%s'\n",net_idx, net_names[net_idx]);
    //get input tensor and output tensor info
    bmrt_get_input_tensor(p_bmrt, net_idx, NULL, &input_tensor_num, &input_tensor_names);
    bmrt_get_output_tensor(p_bmrt, net_idx, NULL, &output_tensor_num, &output_tensor_names);

    //alloc input memory
    int8_t** host_input_data = (int8_t**)(new int8_t* [input_tensor_num]);
    bm_device_mem_t* dev_input_data = new bm_device_mem_t [input_tensor_num];
    int in = 0, ic = 0, ih = 0, iw = 0;
    int* input_dim = new int [input_tensor_num];
    int* input_shape = new int [input_tensor_num * 4];
    for(int i = 0; i < input_tensor_num; ++i) {
      //get the shape of the input tensor
      bmrt_get_input_blob_max_nhw(p_bmrt, input_tensor_names[i], net_idx, NULL, &in, &ic, &ih,␣
↪&iw);
      input_dim[i] = 4;
      input_shape[4*i + 0] = in;
      input_shape[4*i + 1] = ic;
```

(continues on next page)

(continued from previous page)

```
      input_shape[4*i + 2] = ih;
      input_shape[4*i + 3] = iw;
      //get the data type of the input tensor
      int i_dtype = bmrt_get_input_data_type(p_bmrt, input_tensor_names[i], net_idx, NULL);
      //malloc host memory for input data
      host_input_data[i] = new int8_t [in * ic * ih * iw * dtype_size(i_dtype)];
      //read input data from reference
      if(!(fread(host_input_data[i], in * ic * ih * iw * dtype_size(i_dtype), 1, f_input_
↪ref))) {
        printf("Failed to fread reference data for the %d-th input\n", i);
        exit(-1);
      }
      //malloc device memory
      bmrt_malloc_device_byte(p_bmrt, &dev_input_data[i], in * ic * ih * iw * dtype_size(i_
↪dtype));
    }

    //alloc output memory
    int8_t** host_output_data = (int8_t**)(new int8_t* [output_tensor_num]);
    int8_t** ref_output_data = (int8_t**)(new int8_t* [output_tensor_num]);
    bm_device_mem_t* dev_output_data = new bm_device_mem_t [output_tensor_num];
    int* on = new int [output_tensor_num];
    int* oc = new int [output_tensor_num];
    int* oh = new int [output_tensor_num];
    int* ow = new int [output_tensor_num];
    int* o_dtpye = new int [output_tensor_num];
    for(int i = 0; i < output_tensor_num; ++i) {
      //get the shape of the output tensor
      bmrt_get_output_blob_max_nhw(p_bmrt, output_tensor_names[i], net_idx, NULL,
                                   &on[i], &oc[i], &oh[i], &ow[i]);
      //get the data type of the output tensor
      o_dtpye[i] = bmrt_get_output_data_type(p_bmrt, output_tensor_names[i], net_idx, NULL);
      host_output_data[i] = new int8_t [on[i] * oc[i] * oh[i] * ow[i] * dtype_size(o_
↪dtpye[i])];
      ref_output_data[i] = new int8_t [on[i] * oc[i] * oh[i] * ow[i] * dtype_size(o_dtpye[i])];
      //read output data from reference
      if(!(fread(ref_output_data[i], on[i] * oc[i] * oh[i] * ow[i] * dtype_size(o_dtpye[i]),
                 1, f_output_ref))) {
        printf("Failed to fread reference data for the %d-th output\n", i);
        exit(-1);
      }
      //malloc device memory
      bmrt_malloc_device_byte(p_bmrt, &dev_output_data[i],
          on[i] * oc[i] * oh[i] * ow[i] * dtype_size(o_dtpye[i]));
    }

    //memcpy input data from system to device
    for(int i = 0; i < input_tensor_num; ++i) {
      bmrt_memcpy_s2d(p_bmrt, dev_input_data[i], ((void*)host_input_data[i]));
    }

    //neuron network inference
    bool success = false;
    if(bmrt_can_batch_size_change(p_bmrt, net_idx, NULL) ||
       bmrt_can_height_and_width_change(p_bmrt, net_idx, NULL)) {
      //call inference if input shape can be changed
      success = bmrt_launch_shape(p_bmrt, net_idx, NULL, (const void*)(dev_input_data), input_
↪tensor_num, input_dim, input_shape,
                                  (const void*)(dev_output_data), output_tensor_num);
    } else {
      //call inference if input shape cannot changed
```

(continues on next page)

(continued from previous page)

```
      success = bmrt_launch(p_bmrt, net_idx, NULL, (const void*)(dev_input_data), input_tensor_
↪num,
                            (const void*)(dev_output_data), output_tensor_num);
    }

    if(!success) {
      printf("The %d-th neuron network '%s' inference failed\n", net_idx, net_names[net_idx]);
    }

    //sync, wait for finishing inference
    bmrt_thread_sync(p_bmrt);

    //memcpy output data from device to system
    for(int i = 0; i < output_tensor_num; ++i) {
      bmrt_memcpy_d2s(p_bmrt, host_output_data[i], dev_output_data[i]);
    }

    //get true performance time
    long unsigned int last_api_process_time_us = 0;
    bmrt_get_last_api_process_time_us(p_bmrt, &last_api_process_time_us);
    printf("the last_api_process_time_us is %lu us\n", last_api_process_time_us);

    //compare inference output data with reference data
    int flag = result_cmp(host_output_data, ref_output_data, output_tensor_num, on, oc, oh, ow,
↪ o_dtpye);
    if(flag != 0) {
      printf("+++ The %d-th network '%s' cmp failed +++\n", net_idx, net_names[net_idx]);
      exit(-1);
    } else {
      printf("+++ The %d-th network '%s' cmp success +++\n", net_idx, net_names[net_idx]);
    }

    //free tensor name
    free(input_tensor_names);
    free(output_tensor_names);
    //free memory
    delete [] dev_input_data;
    for(int i = 0; i < input_tensor_num; ++i) {
      delete [] host_input_data[i];
    }
    delete [] host_input_data;
    delete [] dev_output_data;
    for(int i = 0; i < output_tensor_num; ++i) {
      delete [] host_output_data[i];
      delete [] ref_output_data[i];
    }
    delete [] host_output_data;
    delete [] ref_output_data;
    delete [] on;
    delete [] oc;
    delete [] oh;
    delete [] ow;
    delete [] o_dtpye;
    delete [] input_dim;
    delete [] input_shape;
  }

  free(net_names);
  fclose(f_input_ref);
  fclose(f_output_ref);
}
```