BMRuntime Technical Reference Manual

SOPHGO

Oct 27, 2025

Contents

CHAPTER 1

Disclaimer

1.1 Disclaimer

Technical Support

Address
Floor 6, Building 1, Yard 9, FengHao East Road, Haidian District, Beijing, 100094, China

Website
https://www.sophgo.com/

Email
sales@sophgo.com

Phone
+86-10-57590723 +86-10-57590724

SDK Release History

Version	Release Date	Description
V2.0.0	2019.09.20	Released the initial version.
V2.0.1	2019.11.16	Released Version V2.0.1.
V2.0.3	2020.05.07	Released Version V2.0.3.
V2.2.0	2020.10.12	Released Version V2.2.0.
V2.3.0	2021.01.11	Released Version V2.3.0.
V2.3.1	2021.03.09	Released Version V2.3.1.
V2.3.2	2021.04.01	Released Version V2.3.2.
V2.4.0	2021.05.23	Released Version V2.4.0.
V2.5.0	2021.09.02	Released Version V2.5.0.
V2.6.0	2021.01.30	Revised and released Version V.2.6.0.
V2.7.0	2024.03.16	Released Version V2.7.0, patch version on May 31, 2024.
V3.0.0	2024.07.16	Released Version V3.0.0.

CHAPTER 2

BModel

2.1 BModel

2.1.1 About BModel

Bmodel is a deep neural network model file format for SOPHON deep-learning processors. Generated by model compilers (such as tpu-mlir, etc.), it contains parameter information of one or more networks, such as input and output information. It is loaded and used as a model file in the runtime phase.

Bmodel also serves as the compilation output file for the BMLang programming language and it is generated in the BMLang compilation phase. Bmodel contains the information of one or more BMLang functions,such as parameters, input and output.

Multi-stage bmodel description:

> In bmodel, stage is to compile bmodels with various types of input_shape, and then use tpu_model to combine several bmodels into one bmodel, and each bmodel contained in it is a stage.Stage_num indicates the number of combined bmodels. Stage_num=1 for bmodels not combined. When running a model with a given shape, bmruntime will automatically choose a bmodel with the same input shape to run.
>
> The running efficiency can be improved and the effect of dynamic operation can be achieved by selecting several common input shapes to combine. For example, compile two bmodels with the inputs being [1,3,200,200] and [2,3,200,200]. Running will start upon combination. If running is realized with the input [2,3,200,200], the bmodel with the input being [2,3,200,200] will be automatically found to run.

Alternatively, compile two bmodels with the inputs [1,3,200,200] and [1,3,100,100] to build the model that supports inputs 200*200 and 100*100.

Static bmodel description:

1. Static bmodel saves the atomic operation instructions with fixed parameters that can be directly used on the device. The deep-learning processor can automatically read such atomic operation instructions, execute them in the flow line without any interruption in the halfway.
2. When the static bmodel is executed, the size input of the model must be same with its size in compilation.
3. Due to the simplicity and stability of a static interface, the model compiled under the new sdk can usually run on the old machine without refreshing the new firmware. It should be noted that although static compilation is designated for some models, some operators must have the internal MCU of deep-learning processor or host processor involved, such as sorting, nms, where, detect_out and other operators with more logical operations. This part will be divided into subnets and implemented in a dynamic way. If the part for updating sdk compilation is a dynamic model, it is preferred to refresh or update the firmware to ensure sdk and runtime are consistent (It can be judged by the output of tpu_model --info xx.bmodel. For static compilation, if the subnet number is 1, it indicates a purely static network. See the section on tpu_model use for details.
4. If the input shape only has several fixed discrete cases, the multi-stage bmodel aforementioned may be used to achieve the effect of the dynamic model.

Dynamic bmodel description:

1. The dynamic bmodel stores the parameter information of each operator and cannot run directly on the deep-learning processor. It is necessary for the MCU inside the deep-learning processor to parse parameters layer by layer for shape and dtype inference and call atomic operations for achieving specific functions. So, the dynamic bmodel has a worse performance than the static one.
2. When running the dynamic bmodel on the bm168x platform, it is preferred to start icache. Otherwise, the bmodel will run slowly.
3. In the case of compilation, specify the maximum shape available through the shapes parameter. In the event of actual running, except for the c-dimension, variability is available in other dimensions. Normally, dynamic compilation should be considered if there are too many variable shapes. Otherwise, the multi-stage bmodel is recommended.
4. In order to ensure the parameters are scalable and compatible, the dynamic bmodel will ensure the runtime of the new sdk can run the dynamic bmodel compiled by the old version sdk. It is usually recommended to refresh the machine after a new sdk is replaced with so as to ensure the consistency of the two versions.

2.1.2 Tpu_model use

With the tpu_model tool, you can view the parameter information of the bmodel file. Decompose multi-network bmodels into multiple single-network bmodels or combine multi-network bmodels into one bmodel.

Currently, the following six operation methods are available:

1. View brief information (commonly used)

   tpu_model --info xxx.bmodel

   The output information is described as follows:

```
bmodel version: B.2.2                         # bmodel version No.
chip: BM1684                                  # Chip type supported
create time: Mon Apr 11 13:37:45 2024          # Creation time

==========================================
# Network dividing line: If there are multiple nets, there will be several dividing lines.
net 0: [informer_frozen_graph]  static        # The network is named informer_frozen_
↪graph, a static type network (or static network)or a dynamic compilation network
# if it is dynamic.
------------                                  # stage dividing line: There will be several
# dividing lines if there are several stages in each network.
stage 0:                                      # Information of the first stage
subnet number: 41                             # Number of subnets in this stage, which is
# divided at the time of compilation to support the switching of different devices.
↪Normally, the fewermore the number of subnets is
                                              # the better the result will be.
input: x_1, [1, 600, 9], float32, scale: 1    # Input and output information:name, shape,
↪ quantified scale value
input: x_3, [1, 600, 9], float32, scale: 1
input: x, [1, 500, 9], float32, scale: 1
input: x_2, [1, 500, 9], float32, scale: 1
output: Identity, [1, 400, 7], float32, scale: 1

device mem size: 942757216 (coeff: 141710112, instruct: 12291552, runtime:
788755552)  # The memory size occupied by the model on the deep-learning processor,
↪ in byte,
# format: Total memory size occupied (size of constant memory, size of instruction
# memory, size of data memory during the running)
host mem size: 8492192 (coeff: 32, runtime: 8492160)   # Memory size occupied on
# the host, in byte, format: Total memory size occupied (size of constant memory,
# size of data memory during the running)
```

2. View detailed parameter information

   tpu_model --print xxx.bmodel

3. Decompose

   tpu_model --extract xxx.bmodel

Decompose a bmodel that includes several stages in several networks into each bmodel that includes a stage within a network. The decomposed bmodel is named bm_net0_stage0.bmodel, bm_net1_stage0.bmodel and so on according to the serial numbers of net and stage.

4. Combine

   ```
   tpu_model --combine a.bmodel b.bmodel c.bmodel -o abc.bmodel
   ```

   Combine multiple bmodels into one bmodel. -o is used to specify the output file name. If not specified, it is named compilation.bmodel by default.

   Upon the combination of multiple bmodels:

   · Combination of bmodels with different net_names:The interface will select the corresponding network for inference according to net_name.

   · Combination of bmodels with the same net_name: The network with the net_name can support multiple stages, that is multiple input shapes. The interface will make a selection among multiple stages in the network according to the shape you input. For a static network, the stage that perfectly matches the shape will be selected. For a dynamic network, the nearest stage will be selected.

   Restrictions:The same network net_name, when using combine, requires all static compilation, or all dynamic compilation. The combine that adopts static and dynamic compilation for the same net_name is not available.

5. Combine folders

   ```
   tpu_model --combine_dir a_dir b_dir c_dir -o abc_dir
   ```

   It is the same with the functions of combine. Differently, this function can also combine input and output files for testing in addition to the bmodel. It combines folders, each of which must contain three files generated by the compiler: input_ref_data.dat, output_ref_data.dat, compilation.bmodel.

6. Export binary data

   ```
   tpu_model --dump xxx.bmodel start_offset byte_size out_file
   ```

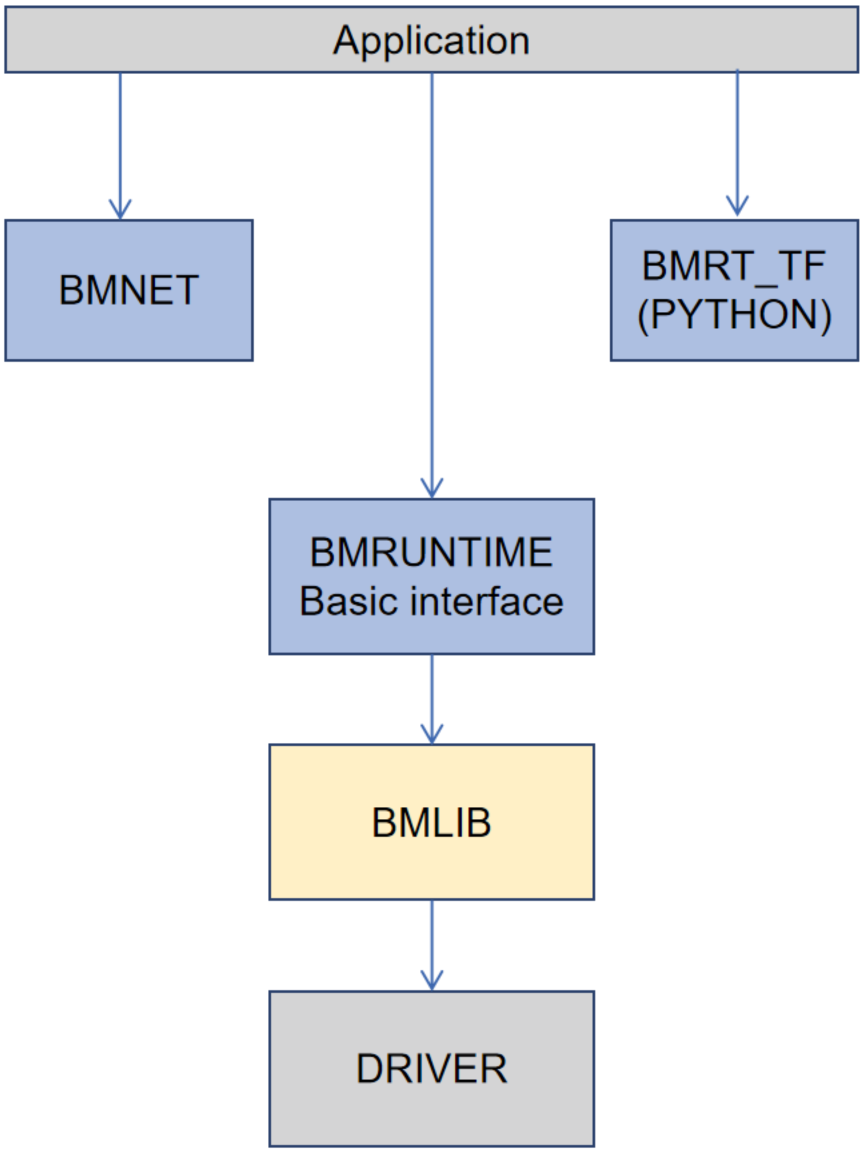
   Save the binary data in bmodel to a file. The print function may be used to view the [start,size] for all binary data, which corresponds to start_offset and byte_size.

CHAPTER 3

BMRuntime

3.1 BMRuntime

BMRuntime is used to read the compiled output (.bmodel) of BMCompiler and drive it to be executed in the deep-learning processor. BMRuntime provides users with diversified interfaces, which are convenient for users to transplant algorithms. Its software architecture is shown as follows:



BMRuntime has two interfaces available, C and C++. Some interfaces are reserved in order to be compatible with the previous generation of applications. However, it is not recommened to continue using new applications.

Interfaces in this chapter are all synchronous by default and some of them are asynchronous (functions are executed by the deep-learning processor and the host processor may continue to execute from up to bottom), which will be specially described.

This chapter consists of four parts:

- BMLIB interface: For device management but not classified into BMRuntime. The interface needs to be combined with others. So, it is introduced at the very beginning.
- C interface: C language interface of BMRuntime
- C++ interface: C++ language interface of BMRuntime

- Multi-thread programming: Introduce how to use the C or C++ interface for multi-thread programming

3.1.1 BMLIB Interface

The BMLIB interface is a C interface, with the corresponding header file being bmlib_runtime.h and the corresponding lib library being libbmlib.so.

The BMLIB interface is used for device management, including device memory management.

There are many BMLIB interfaces. Interfaces that are usually needed by applications are introduced here.

Device

bm_dev_request

```
/* [out] handle
 * [in]  devid
 */
bm_status_t bm_dev_request(bm_handle_t *handle, int devid);
```

Request a device and get the handle of the device. For other device interfaces (bm_xxxx class interfaces), this device handle needs to be specified.

Where, devid means the device No.. Under the PCIE mode, the corresponding device can be selected when there are multiple devices. Please specify 0 under the SoC mode.

In the case of successful request, return BM_SUCCESS; otherwise, return other error codes.

bm_dev_free

```
 /* [out] handle
 */
void bm_dev_free(bm_handle_t handle);
```

Release a device. Normally, an application needs to request a device at the beginning and release such device before its exit.

The reference is shown as follows:

```
// start program
bm_handle_t bm_handle;
bm_dev_request(&bm_handle, 0);
// do things here
......
// end of program
bm_dev_free(bm_handle);
```

Device Memory

bm_malloc_device_byte

```
/* [in]  handle
 * [out] pmem
 * [in]  size
 */
bm_status_t bm_malloc_device_byte(bm_handle_t handle, bm_device_mem_t *pmem,
                                  unsigned int size);
```

Request a device mem of the specified size, where size is the size of the device mem, in byte.

In the case of a successful application, return BM_SUCCESS; otherwise, return other error codes.

bm_free_device

```
/* [in]  handle
 * [out] mem
 */
void bm_free_device(bm_handle_t handle, bm_device_mem_t mem);
```

Relase device mem. Any device mem requested will need to be released when it is not used any more.

The reference is shown as follows:

```
// alloc 4096 bytes device mem
bm_device_mem_t mem;
bm_status_t status = bm_malloc_device_byte(bm_handle, &mem, 4096);
assert(status == BM_SUCCESS);
// do things here
......
// if mem will not use any more, free it
bm_free_device(bm_handle, mem);
```

bm_mem_get_device_size

```
// [in] mem
unsigned int bm_mem_get_device_size(struct bm_mem_desc mem);
```

Get the size of device mem, in byte.

bm_memcpy_s2d

Copy data on system memory to device mem. System memory is specified by the void pointer and device men is specified the by bm_device_mem_t type.

In the case of successful copying, return BM_SUCCESS; otherwise, return other error codes.

There are three types depending on the size and offset of the copy:

```
// The size of the copy is the size of device mem, with the copying starting from src.
/* [in]  handle
 * [out] dst
 * [in]  src
 */
bm_status_t bm_memcpy_s2d(bm_handle_t handle, bm_device_mem_t dst, void *src);
```

```
// size specifies the size of the copy, in byte,and start copying from the offset of src.
/* [in]  handle
 * [out] dst
 * [in]  src
 * [in]  size
 * [in]  offset
 */
bm_status_t bm_memcpy_s2d_partial_offset(bm_handle_t handle, bm_device_mem_t dst,
                                         void *src, unsigned int size,
                                         unsigned int offset);
```

```
// size specifies the size of the copy, in byte, with the copying starting from src.
/* [in]  handle
 * [out] dst
 * [in]  src
 * [in]  size
 */
bm_status_t bm_memcpy_s2d_partial(bm_handle_t handle, bm_device_mem_t dst,
                                  void *src, unsigned int size);
```

bm_memcpy_d2s

Copy data in device mem to the system memory. In the case of successful copying, return BM_SUCCESS; otherwise, return other error codes.

System memory is specified by the void pointer and device mem is specified by the bm_device_mem_t type.

There are three types depending on the size and offset of the copy:

```
// The size of the copy is the size of device mem, with the copying starting from the 0 offset of
// device mem.
/* [in]  handle
```

(continues on next page)

(continued from previous page)

```
 * [out] dst
 * [in]  src
 */
bm_status_t bm_memcpy_d2s(bm_handle_t handle, void *dst, bm_device_mem_t src);
```

```
//size specifies the size of the copy, in byte, with the copying starting from the offset of device
//mem.
/* [in]  handle
 * [out] dst
 * [in]  src
 * [in]  size
 * [in]  offset
 */
bm_status_t bm_memcpy_d2s_partial_offset(bm_handle_t handle, void *dst,
                                          bm_device_mem_t src, unsigned int size,
                                          unsigned int offset);
```

```
// size specifies the size of the copy, in byte, with the copying starting from the 0 offset of device
// mem.
/* [in]  handle
 * [out] dst
 * [in]  src
 * [in]  size
 */
bm_status_t bm_memcpy_d2s_partial(bm_handle_t handle, void *dst,
                                   bm_device_mem_t src, unsigned int size);
```

bm_memcpy_d2d

```
/* [in]  handle
 * [out] dst
 * [in]  dst_offset
 * [in]  src
 * [in]  src_offset
 * [in]  len
 */
bm_status_t bm_memcpy_d2d(bm_handle_t handle, bm_device_mem_t dst, int dst_offset,
                           bm_device_mem_t src, int src_offset, int len);
```

Copy data from a device mem to the other.

dst_offset specifies the offset of the target; src_offset specifies the offset of the source; and len specifies the size of the copy.

Special notice: len is in dword. For example, to copy 1024 bytes, len needs to be specified as 1024/4=256.

Device Memory Mmap

The interface described in this section is only valid on the SoC. On the SoC, although the system memory and Device Memory are separated from each other, they are actually memories on DDR.

Mmap may be used to get the virtual address of Device Memory so that it can be directly accessed by the application.

Special note: The deep-learning processor directly accesses DDR when accessing Device Memory without passing cache but cache is passed when an application accesses it.

Thus, it is necessary to ensure the consistency of caches. This means:

- The application revises the data of Device Memory through the virtual address. It is necessary to flush before deep-learning processor inference so as to ensure the cache data has been synchronized with DDR.
- Device Memory data is modified upon the ending of deep-learning processor inference. The application needs to be invalidated before access through the virutal address so as to ensure DDR data has been synchronized with cache.

bm_mem_mmap_device_mem

```
/* [in]  handle
 * [in]  dmem
 * [out] vmem
 */
bm_status_t bm_mem_mmap_device_mem(bm_handle_t handle,
                                   bm_device_mem_t *dmem,
                                   unsigned long long *vmem);
```

Map device mem and get a virtual address.

Return BM_SUCCESS if it is successful; otherwise, return other error codes.

bm_mem_unmap_device_mem

```
/* [in]  handle
 * [out] vmem
 * [in]  size
 */
bm_status_t bm_mem_unmap_device_mem(bm_handle_t handle,
                                    void* vmem, int size);
```

Unmap is required when the mapped virtual address is not used any more. Size indicates the size of device mem, which can be obtained through bm_mem_get_device_size.

bm_mem_invalidate_device_mem

```
/* [in]  handle
 * [in]  dmem
 */
bm_status_t bm_mem_invalidate_device_mem(bm_handle_t handle, bm_device_mem_t
*dmem);
```

To invalidate cache is to ensure DDR data is sychronized with the cache.

```
/* [in]  handle
 * [out] dmem
 * [in]  offset
 * [in]  len
 */
bm_status_t bm_mem_invalidate_partial_device_mem(bm_handle_t handle, bm_device_
↪mem_t *dmem,unsigned int offset, unsigned int len);
```

Specify that cache is invalidated within the offset and size of device mem.

bm_mem_flush_device_mem

```
/* [in]  handle
 * [out] dmem
 */
bm_status_t bm_mem_flush_device_mem(bm_handle_t handle, bm_device_mem_t *dmem);
```

Refresh cache data or ensure cache data has been sychronized with DDR.

```
/* [in]  handle
 * [out] dmem
 * [in]  offset
 * [in]  len
 */
bm_status_t bm_mem_flush_partial_device_mem(bm_handle_t handle, bm_device_mem_t
*dmem,unsigned int offset, unsigned int len);
```

Specify cache refreshing within the offset and size of device mem.

example

Here is an example of mmap interface use:

```
bm_device_mem_t input_mem, output_mem;
bm_status_t status = bm_malloc_device_byte(bm_handle, &input_mem, 4096);
assert(status == BM_SUCCESS);
status = bm_malloc_device_byte(bm_handle, &output_mem, 256);
```

(continues on next page)

(continued from previous page)

```
assert(status == BM_SUCCESS);
void *input, * output;

// mmap device mem to virtual addr
status = bm_mem_mmap_device_mem(bm_handle, &input_mem, (uint64_t*)&input);
assert(status == BM_SUCCESS);
status = bm_mem_mmap_device_mem(bm_handle, &output_mem, (uint64_t*)&output);
assert(status == BM_SUCCESS);

// copy input data to input， and flush it
memcpy(input, input_data, 4096);
status = bm_mem_flush_device_mem(bm_handle, &input_mem);
assert(status == BM_SUCCESS);

// do inference here
......

// invalidate output, and copy output data from output
status = bm_mem_invalidate_device_mem(bm_handle, &output_mem);
assert(status == BM_SUCCESS);
memcpy(output_data, output, 256);

// unmap
status = bm_mem_unmap_device_mem(bm_handle, input, 4096);
assert(status == BM_SUCCESS);
status = bm_mem_unmap_device_mem(bm_handle, output, 256);
assert(status == BM_SUCCESS);
```

Program synchronize

```
// [in] handle
bm_status_t bm_thread_sync(bm_handle_t handle);
```

Synchronous interface. Normally, deep-learning processor inference is made asynchronously and the user’ s host program can continue to be executed. This interface is used in the host process to ensure the deep-learning processor inference is completed. Unless otherwise specially described, all interfaces introduced in this chapter are synchronous ones. There are only a few asynchronous interfaces that need to call bm_thread_sync for synchronization.

3.1.2 C Interface

The C interface of BMRuntime, with the corresponding header file being bmruntime_interface.h and the corresponding lib library being libbmrt.so.

It is recommended to use this interface when the user’ s program uses the C interface, which supports static compilation networks in various shapes and dynamic compilation networks.

Tensor information

Tensor represents multi-dimensional data and the data operated in BMRuntime is Tensor.

Data type

```
typedef enum bm_data_type_e {
  BM_FLOAT32 = 0,
  BM_FLOAT16 = 1,
  BM_INT8 = 2,
  BM_UINT8 = 3,
  BM_INT16 = 4,
  BM_UINT16 = 5,
  BM_INT32 = 6,
  BM_UINT32 = 7
} bm_data_type_t;
```

bm_data_type_t is used to indicate the data type.

Store mode

```
/* store mode definitions */
typedef enum bm_store_mode_e {
  BM_STORE_1N = 0, /* default, if not sure, use 0 */
  BM_STORE_2N = 1,
  BM_STORE_4N = 2,
} bm_store_mode_t;
```

bm_store_mode_t specifies how data is stored. You only need to focus on BM_STORE_1N. If you want to focus on the bottom layer and optimize performance, you need to focus on BM_STORE_2N and BM_STORE_4N.

BM_STORE_1N is the default storage method for data types. It indicates data is stored as normal.

BM_STORE_2N is only used for BM_FLOAT16/BM_INT16/BM_UINT16. It indicates the data with two different batches and the same other dimension positions are placed in a 32-bit data space. For example, for a four-dimensional (n, c, h, w) tensor, (0, ci, hi, wi) data is placed in the lower 16 bits of 32 bits and (1, ci, hi, wi) is placed in the upper 16 bits.

BM_STORE_4N is only used for BM_INT8/BM_UINT8. It indicates that the data with four different batches and the same other dimension positions are placed in a 32-bit data space. For example, for a four-dimensional (n, c, h, w) tensor, (0, ci, hi, wi) data is placed in the 0 to 7 bits of 32 bits, (1, ci, hi, wi) data is placed in the 8 to 15 bits, (2, ci, hi, wi) data is placed in the 16 to 23 bits and (3, ci, hi, wi) data is placed in the 24 to 31 bits.

Shape

```
/* bm_shape_t holds the shape info */
#define BM_MAX_DIMS_NUM 8
typedef struct bm_shape_s {
  int num_dims;
  int dims[BM_MAX_DIMS_NUM];
} bm_shape_t;
```

bm_shape_t represents the shape of tensor, with the tensor of up to eight dimensions supported. Where, num_dims represents the number of dimensions for the tensor; dims represents the value of each dimension, with each dimension value of dims starting from [0]. For example, the four dimensions (n, c, h, w) correspond to (dims [0], dims[1], dims[2], dims[3]) respectively.

In the case of constant shape, the initialization reference is shown as follows:

```
bm_shape_t shape = {4, {4,3,228,228}};
bm_shape_t shape_array[2] = {
       {4, {4,3,28,28}}, // [0]
       {2, {2,4}}, // [1]
};
```

The bmrt_shape interface is used to set bm_shape_t as follows:

```
/*
dims array to bm_shape_t,
shape and dims should not be NULL, num_dims should not be larger than BM_MAX_DIMS_
↪NUM

Parameters: [out] shape   - The bm_shape_t pointer.
            [in] dims     - The dimension value.
                            The sequence is the same with dims[BM_MAX_DIMS_NUM].
            [in] num_dims - The number of dimension.
*/
void bmrt_shape(bm_shape_t* shape, const int* dims, int num_dims);
```

bmrt_shape_count can be used to get the number of shape elements. The interface is declared as follows:

```
/*
number of shape elements, shape should not be NULL and num_dims should not large than
BM_MAX_DIMS_NUM */
uint64_t bmrt_shape_count(const bm_shape_t* shape);
```

For example, if num_dims is 4, the number of dims got is dims[0]*dims[1]*dims[2]*dims[3]. If num_dims is 0, return 1.

The bmrt_shape_is_same interface is used to judge if two shapes are the same. The interface is declared as follows:

```
/* compare whether two shape is same */
bool bmrt_shape_is_same(const bm_shape_t* left, const bm_shape_t* right);
```

Return "true" if two shapes are the same and "false" if they are different.

The interface is considered to be of the same shape only if num_dims and the corresponding dims[0], dims[1], ⋯dims[num_dims-1] are the same.

Tensor

The bm_tensor_t structure is used to represent a tensor:

```
/*
bm_tensor_t holds a multi-dimensional array of elements of a single data type
and tensor are in device memory */
typedef struct bm_tensor_s {
  bm_data_type_t dtype;
  bm_shape_t shape;
  bm_device_mem_t device_mem;
  bm_store_mode_t st_mode; /* user can set 0 as default store mode */
} bm_tensor_t;
```

The bmrt_tensor can be configured with a tensor. The interface is declared as follows:

```
/*
This API is to initialize the tensor. It will alloc device mem to tensor->device_mem,
so user should use bm_free_device(p_bmrt, tensor->device_mem) to free it.
After initialization, tensor->dtype = dtype, tensor->shape = shape, and tensor->st_mode = 0.

Parameters: [out] tensor - The pointer of bm_tensor_t. It should not be NULL.
            [in]  p_bmrt - The pointer of bmruntime. It should not be NULL
            [in]  dtype  - The data type.
            [in]  shape  - The shape.
*/
void bmrt_tensor(bm_tensor_t* tensor, void* p_bmrt, bm_data_type_t dtype, bm_shape_t
shape);
```

The bmrt_tensor_with_device interface is used to configure a tensor with the existing device mem. The interface is declared as follows:

```
/*
The API is to initialize the tensor with a existed device_mem.
The tensor byte size should not be larger than device mem size.
After initialization, tensor->dtype = dtype, tensor->shape = shape,
```

(continues on next page)

(continued from previous page)

```
tensor->device_mem = device_mem, and tensor->st_mode = 0.

Parameters: [out] tensor     - The pointer of bm_tensor_t. It should not be NULL.
            [in]  device_mem - The device memory that had be allocated device memory.
            [in]  dtype      - The data type.
            [in]  shape      - The shape.
*/
void bmrt_tensor_with_device(bm_tensor_t* tensor, bm_device_mem_t device_mem,
                             bm_data_type_t dtype, bm_shape_t shape);
```

Here, the bmrt_tensor and bmrt_tensor_with_device interfaces are used to provide convenience for you to initialize a tensor.You can also initialize each member of bm_tensor_t without the aid of any interface.

bmrt_tensor_bytesize is used to get the size of tensor and is measured in byte. It is obtained by multiplying the number of tensor elements by the number of bytes for the data type. The interface is declared as follows:

```
/*
Parameters: [in] tensor - The pointer of bm_tensor_t. It should not be NULL.
Returns:    size_t      - The byte size of the tensor.
*/
size_t bmrt_tensor_bytesize(const bm_tensor_t* tensor);
```

bmrt_tensor_device_size is used to get the size of device mem, in byte. The interface is declared as follows:

```
/*
Parameters: [in] tensor - The pointer of bm_tensor_t. It should not be NULL.
Returns:    size_t      - The byte size of the tensor->dev_mem.
*/
size_t bmrt_tensor_device_size(const bm_tensor_t* tensor);
```

bmrt_create

```
/*
Parameters: [in] bm_handle - BM handle. It must be declared and initialized by using bmlib.
Returns:    void*          - The pointer of a bmruntime helper.
*/
void* bmrt_create(bm_handle_t bm_handle);
```

Create bmruntime and return the runtime pointer. For other interfaces (bmrt_xxxx class interfaces), the required handle is the runtime pointer.

bmrt_create_ex

```
/*
Parameters: [in] bm_handles   - BM handles. They must be initialized by using bmlib.
Parameters: [in] num_handles  - Number of bm_handles.
Returns:    void*             - The pointer of a bmruntime helper.
*/
void *bmrt_create_ex(bm_handle_t *bm_handles, int num_handles);
```

Create a bmruntime that supports passing in multiple bm_handle, used to run distributed bmodels.

bmrt_destroy

```
/*
Parameters: [in] p_bmrt - Bmruntime helper that had been created.
*/
void bmrt_destroy(void* p_bmrt);
```

Destroy bmruntime and release resources.

Normally, the user starts to create runtime and destroy runtime before exit. The example is shown as follows:

```
// start program
bm_handle_t bm_handle;
bm_dev_request(&bm_handle, 0);
void * p_bmrt = bmrt_create(bm_handle);
// do things here
......
// end of program
bmrt_destroy(p_bmrt);
bm_dev_free(bm_handle);
```

bmrt_get_bm_handle

```
/*
Parameters: [in]  p_bmrt   - Bmruntime that had been created
Returns:    void*          - The pointer of bm_handle_t
*/
void * bmrt_get_bm_handle(void* p_bmrt);
```

Get bm_handle, the handle of the device, from the runtime pointers. The handle is required by bm_xxxx class interfaces.

bmrt_load_bmodel

```
/*
Parameters: [in] p_bmrt       - Bmruntime that had been created.
            [in] bmodel_path - Bmodel file directory.
Returns:    bool              - true: success; false: failed.
*/
bool bmrt_load_bmodel(void* p_bmrt, const char *bmodel_path);
```

Load the bmodel file. Upon loading, there will be data of several networks in bmruntime. The networks may be subsequently inferred.

bmrt_load_bmodel_data

```
/*
Parameters: [in] p_bmrt       - Bmruntime that had been created.
            [in] bmodel_data - Bmodel data pointer to buffer.
            [in] size        - Bmodel data size.
Returns:    bool              - true: success; false: failed.
*/
bool bmrt_load_bmodel_data(void* p_bmrt, const void * bmodel_data, size_t size);
```

Load bmodel. Different from bmrt_load_bmodel, its bmodel data is stored in the memory.

bmrt_show_neuron_network

```
/*
Parameters: [in] p_bmrt - Bmruntime that had been created.
*/
void bmrt_show_neuron_network(void* p_bmrt);
```

Print the names of networks in bmruntime.

bmrt_get_network_number

```
/*
Parameters: [in] p_bmrt - Bmruntime that had been created
Returns:    int          - The number of neuron networks.
*/
int bmrt_get_network_number(void* p_bmrt);
```

Get the number of networks in bmruntime.

bmrt_get_network_names

```
/*
Parameters:[in]  p_bmrt        - Bmruntime that had been created.
         [out] network_names - The names of all neuron networks.

Note:
network_names should be declared as (const char** networks = NULL), and use as &networks.
After this API, user need to free(networks) if user do not need it.
*/
void bmrt_get_network_names(void* p_bmrt, const char*** network_names);
```

Get the names of all networks in the runtime. This interface will request the memory for network_names. So, the interface needs to call free to release when it is not used any more.

The example of the use method is shown as follows:

```
const char **net_names = NULL;
int net_num = bmrt_get_network_number(p_bmrt);
bmrt_get_network_names(p_bmrt, &net_names);
for (int i=0; i<net_num; i++) {
  puts(net_names[i]);
}
free(net_names);
```

bmrt_get_network_info

Network information is expressed as follows:

```
/* bm_stage_info_t holds input shapes and output shapes;
every network can contain one or more stages */
typedef struct bm_stage_info_s {
  bm_shape_t* input_shapes;   /* input_shapes[0] / [1] / ... / [input_num-1] */
  bm_shape_t* output_shapes;  /* output_shapes[0] / [1] / ... / [output_num-1] */
  bm_device_mem_t *input_mems; /* input_mems[0] / [1] / ... / [input_num-1] */
  bm_device_mem_t *output_mems; /* output_mems[0] / [1] / ... / [output_num-1] */
} bm_stage_info_t;

/* bm_tensor_info_t holds all information of one net */
typedef struct bm_net_info_s {
  const char* name;              /* net name */
  bool is_dynamic;               /* dynamic or static */
  int input_num;                 /* number of inputs */
  char const** input_names;      /* input_names[0] / [1] / .../ [input_num-1] */
  bm_data_type_t* input_dtypes;  /* input_dtypes[0] / [1] / .../ [input_num-1] */
  float* input_scales;           /* input_scales[0] / [1] / .../ [input_num-1] */
  int output_num;                /* number of outputs */
  char const** output_names;     /* output_names[0] / [1] / .../ [output_num-1] */
  bm_data_type_t* output_dtypes; /* output_dtypes[0] / [1] / .../ [output_num-1] */
  float* output_scales;          /* output_scales[0] / [1] / .../ [output_num-1] */
```

(continues on next page)

(continued from previous page)

```
  int stage_num;                  /* number of stages */
  bm_stage_info_t* stages;        /* stages[0] / [1] / ... / [stage_num-1] */
  size_t * max_input_bytes;       /* max_input_bytes[0]/ [1] / ... / [input_num-1] */
  size_t * max_output_bytes;      /* max_output_bytes[0] / [1] / ... / [output_num-1] */
  int* input_zero_point;          /* input_zero_point[0] / [1] / .../ [input_num-1] */
  int* output_zero_point;         /* output_zero_point[0] / [1] / .../ [output_num-1] */
  int *input_loc_devices;         /* input_loc_device[0] / [1] / .../ [input_num-1] */
  int *output_loc_devices;        /* output_loc_device[0] / [1] / .../ [output_num-1] */
  int core_num;                   /* core number */
  int32_t addr_mode;              /* address assign mode */
} bm_net_info_t;
```

bm_net_info_t represents all information of a network and bm_stage_info_t represents the conditions of different shapes supported by the network.

input_num represents the number of inputs, input_names/input_dtypes/input_scales and input_shapes in bm_stage_info_t indicates this number.

output_num represents the number of outputs,output_names/output_dtypes/output_scales and output_shapes in bm_stage_info_t indicates this number.

input_scales and output_scales are only useful when they are integers and are 1.0 by default when they are of float type.

max_input_bytes represents the maximum number of bytes for each input and max_output_bytes represents the maximum number of bytes for each output. Each network may have multiple stages. The user may request the maximum number of bytes for each input/output and store the data of various stages.

input_zero_point and output_zero_point record the zero_point values for inputs and outputs in the case of an asymmetric quantized int8 network.

input_loc_devices and output_loc_devices record the device id for inputs and outputs in the case of a distributed network.

core_num records the number of cores required by the network.

addr_mode records the network’s address allocation mode, where 0 indicates the basic mode, 1 indicates the io_alone mode, 2 indicates the io_tag mode, and 3 indicates the io_tag_fuse mode.

bmrt_get_network_info gets the information of a given network according to the network name. The interface is declared as follows:

```
/*
Parameters: [in] p_bmrt   - Bmruntime that had been created.
            [in] net_name - Network name.
Returns:    bm_net_info_t - The pointer of bm_net_info_t. If net not found, will return NULL.
*/
const bm_net_info_t* bmrt_get_network_info(void* p_bmrt, const char* net_name);
```

bmrt_print_network_info

Print network information. It is required in debugging. The interface is declared as follows:

```
void bmrt_print_network_info(const bm_net_info_t* net_info);
```

bmrt_launch_tensor

Infer deep-learning processor for the designated network. The interface is declared as follows:

```
/*
To launch the inference of the neuron network with setting input tensors.
This API supports the neuron nework, that is static-compiled or dynamic-compiled.
After calling this API, inference on deep-learning processor is launched. The host program will
↪not be blocked
if the neuron network is static-compiled and has no cpu layer. Otherwize, the host
program will be blocked. This API support multiple inputs, and multi thread safety.

Parameters: [in] p_bmrt - Bmruntime that had been created.
            [in] net_name - The name of the neuron network.
            [in] input_tensors - Array of input tensor.
                                 Defined like bm_tensor_t input_tensors[input_num].
                                 User should initialize each input tensor.
            [in] input_num - Input number.
            [out] output_tensors - Array of output tensor.
                                   Defined like bm_tensor_t output_tensors[output_num].
                                   Data in output_tensors device memory use BM_STORE_1N.
            [in] output_num - Output number.
Returns:    bool - true: Launch success. false: Launch failed.

Note:
This interface will alloc devcie mem for output_tensors. User should free each device mem by
bm_free_device after the result data is useless.
*/
bool bmrt_launch_tensor(void* p_bmrt, const char * net_name,
                        const bm_tensor_t input_tensors[], int input_num,
                        bm_tensor_t output_tensors[], int output_num);
```

The user needs to initialize the input_tensors required by the network before inference, including data in input_tensors. Output_tensors is used to return the inference result.

Special note:

- This interface will request device mem for output_tensors to store result data. You should actively release device mem when you do not need any result data.
- Upon the completion of inference, output data is stored in the form of BM_STORE_1N and the output shape is stored in the shape of each output_tensor .
- This interface is asynchronous. You need to call bm_thread_sync to ensure the inference is completed.

The example of the use method is shown as follows:

```
bm_status_t status = BM_SUCCESS;
bm_tensor_t input_tensors[1];
bm_tensor_t output_tensors[2];
bmrt_tensor(&input_tensors[0], p_bmrt, BM_FLOAT32, {4, {1, 3, 28, 28}});
bm_memcpy_s2d_partial(bm_handle, input_tensors[0].device_mem, (void *)input0,
                      bmrt_tensor_bytesize(&input_tensors[0]));
bool ret = bmrt_launch_tensor(p_bmrt, "PNet", input_tensors, 1, output_tensors, 2);
assert(true == ret);
status = bm_thread_sync(bm_handle);
assert(status == BM_SUCCESS);
bm_memcpy_d2s_partial(bm_handle, output0, output_tensors[0].device_mem,
                      bmrt_tensor_bytesize(&output_tensors[0]));
bm_memcpy_d2s_partial(bm_handle, output1, output_tensors[1].device_mem,
                      bmrt_tensor_bytesize(&output_tensors[1]));
bm_free_device(bm_handle, output_tensors[0].device_mem);
bm_free_device(bm_handle, output_tensors[1].device_mem);
bm_free_device(bm_handle, intput_tensors[0].device_mem);
```

bmrt_launch_tensor_ex

Infer deep-learning processor for a given network. The interface is declared as follows:

```
/*
To launch the inference of the neuron network with setting input tensors.
This API supports the neuron nework, that is static-compiled or dynamic-compiled.
After calling this API, inference on deep-learning processor is launched. The host program will
↪not be blocked
if the neuron network is static-compiled and has no cpu layer. Otherwize, the host
program will be blocked. This API supports multiple inputs, and multi thread safety.

Parameters: [in] p_bmrt - Bmruntime that had been created.
            [in] net_name - The name of the neuron network.
            [in] input_tensors - Array of input tensor.
                                 Defined like bm_tensor_t input_tensors[input_num].
                                 User should initialize each input tensor.
            [in] input_num - Input number.
            [out] output_tensors - Array of output tensor.
                                   Defined like bm_tensor_t output_tensors[output_num].
                                   User can set device_mem or stmode of output tensors.
                                   If user_mem is true, this interface will use device mem of
                                   output_tensors, and will not alloc device mem; Or this
                                   interface will alloc devcie mem to store output.
                                   User should free each device mem by bm_free_device after
                                   the result data is useless.
            [in] output_num - Output number.
            [in] user_mem - true: device_mem in output_tensors have been allocated.
                            false: have not been allocated.
            [in] user_stmode - true: output will use store mode that set in output_tensors.
                               false: output will use BM_STORE_1N.
```

(continues on next page)

(continued from previous page)

```
Returns:    bool - true: Launch success. false: Launch failed.
*/
bool bmrt_launch_tensor_ex(void* p_bmrt, const char * net_name,
                           const bm_tensor_t input_tensors[], int input_num,
                           bm_tensor_t output_tensors[], int output_num,
                           bool user_mem, bool user_stmode);
```

You may specify the output device mem and store mode in output_tensors, which is different from bmrt_launch_tensor.

bmrt_luanch_tensor == bmrt_launch_tensor_ex(user_mem = false, user_stmode = false)

The specific description is as follows:

- When user_mem is false, the interface will request device mem for each output_tensor and save output data.
- When user_mem is true, the interface will not request device mem for output_tensor. You need to make a request from the outside. The size requested can be specified by max_output_bytes in bm_net_info_t.
- When user_stmode is false, the output data is arranged in the form of BM_STORE_1N.
- When user_stmode is true, the output data will be specified according to st_mode in each output_tensor.

Special note: This interface is asynchronous. You need to call bm_thread_sync to ensure the inference is completed.

The example of the use method is shown as follows:

```
bm_status_t status = BM_SUCCESS;
bm_tensor_t input_tensors[1];
bm_tensor_t output_tensors[2];
auto net_info = bmrt_get_network_info(p_bmrt, "PNet");
status = bm_malloc_device_byte(bm_handle, &input_tensors[0].device_mem,
                               net_info->max_input_bytes[0]);
assert(status == BM_SUCCESS);
input_tensors[0].dtype = BM_FLOAT32;
input_tensors[0].st_mode = BM_STORE_1N;
status = bm_malloc_device_byte(bm_handle, &output_tensors[0].device_mem,
                               net_info->max_output_bytes[0]);
assert(status == BM_SUCCESS);
status = bm_malloc_device_byte(bm_handle, &output_tensors[1].device_mem,
                               net_info->max_output_bytes[1]);
assert(status == BM_SUCCESS);

input_tensors[0].shape = {4, {1, 3, 28, 28}};
bm_memcpy_s2d_partial(bm_handle, input_tensors[0].device_mem, (void *)input0,
                      bmrt_tensor_bytesize(&input_tensors[0]));
bool ret = bmrt_launch_tensor_ex(p_bmrt, "PNet", input_tensors, 1,
```

(continues on next page)

(continued from previous page)

```
                           output_tensors, 2, true, false);
assert(true == ret);
status = bm_thread_sync(bm_handle);
assert(status == BM_SUCCESS);
bm_memcpy_d2s_partial(bm_handle, output0, output_tensors[0].device_mem,
                    bmrt_tensor_bytesize(&output_tensors[0]));
bm_memcpy_d2s_partial(bm_handle, output1, output_tensors[1].device_mem,
                    bmrt_tensor_bytesize(&output_tensors[1]));
bm_free_device(bm_handle, output_tensors[0].device_mem);
bm_free_device(bm_handle, output_tensors[1].device_mem);
bm_free_device(bm_handle, intput_tensors[0].device_mem);
```

bmrt_launch_data

Infer deep-learning processor for a given network. The interface is declared as follows:

```
/*
To launch the inference of the neuron network with setting input datas in system memory.
This API supports the neuron nework, that is static-compiled or dynamic-compiled.
After calling this API, inference on deep-learning processor is launched. And the host program
↪will be blocked.
This API supports multiple inputs, and multi thread safety.

Parameters: [in] p_bmrt       - Bmruntime that had been created.
          [in] net_name     - The name of the neuron network.
          [in] input_datas  - Array of input data.
                             Defined like void * input_datas[input_num].
                             User should initialize each data pointer as input.
          [in] input_shapes - Array of input shape.
                             Defined like bm_shape_t input_shapes[input_num].
                             User should set each input shape.
          [in] input_num    - Input number.
          [out]output_datas - Array of output data.
                             Defined like void * output_datas[output_num].
                             If user doesn't alloc each output data, set user_mem to false,
                             and this api will alloc output mem, user should free each
                             output mem when output data not used. Also user can alloc
                             system memory for each output data by self and set user_mem
                             true. Data in memory use BM_STORE_1N.
          [out]output_shapes- Array of output shape.
                             Defined like bm_shape_t output_shapes[output_num].
                             It will store each output shape.
          [in] output_num   - Output number.
          [in] user_mem     - true: output_datas[i] has been allocated memory.
                             false: output_datas[i] has not been allocated memory.
Returns:    bool - true: Launch success; false: Launch failed.
*/
bool bmrt_launch_data(void* p_bmrt, const char* net_name, void* const input_datas[],
                  const bm_shape_t input_shapes[], int input_num, void * output_datas[],
```

(continues on next page)

(continued from previous page)

```
                    bm_shape_t output_shapes[], int output_num, bool user_mem);
```

The difference with bmrt_launch_tensor is as follows:

- Both input and output are stored in the system memory.
- It is a synchronous interface. The inference has been completed when the interface returns.

bmrt_trace

```
/*
To check runtime environment, and collect info for DEBUG.

Parameters: [in] p_bmrt - Bmruntime helper that had been created.
*/
void bmrt_trace(void* p_bmrt);
```

This interface is used for debugging. It can check runtime data and print some information about runtime to faciliate debugging.

3.1.3 C++ Interface

C++ interface of BMRuntime, with the corresponding header file being bmruntime_cpp.h and the corresponding lib library being libbmrt.so. You are suggested to use this interface when using the C++ interface, which supports static compilation networks of multiple shapes and dynamic compilation networks.

The C++ interface naming space is called bmruntime, which consists of three classes and global APIs:

- class Context : Used for network management, it includes loading network models and obtaining network information.
- class Network: It is used to infer a specific network in class Context.
- class Tensor: Automatically generated by class Network, it is used to manage input tensors and output sensors.
- Global APIs: It is used for obtaining the byte size of tensors and the number of elements and comparing whether shapes are identical.

The declaration is as follows:

```
namespace bmruntime {
    class Context;
    class Network;
    class Tensor;
```

(continues on next page)

(continued from previous page)

```
    ......
}
```

class Context

Context is used for network management, such as loading models, which can be loaded from one to multiple models; obtaining network information to get the names of all networks loaded and the information of a given network through network names.

Constructor and destructor

```
explicit Context(int devid = 0);
explicit Context(bm_handle_t bm_handle);
virtual ~Context();
```

Constructor and destructor of Context

When calling the C++ interface, create a Context instance first to specify devid to create an example. The device number 0 is used by default.

The use reference is shown as follows:

```
int main() {
  // start program
  Context ctx;
  // do things here
  ......
  // end of program
}
```

You can also load bm_handle to create an example. Where, bm_handle is generated by bm_dev_request. It should be noted that when the program is exited in this way, Context is first destructed and then bm_dev_free is called to release the bm_handle.

The use reference is as follows:

```
int main() {
  // start program
  bm_handle_t bm_handle;
  bm_dev_request(&bm_handle, 0);
  Context * p_ctx = new Context(bm_handle);
  // do things here
  ......
  // end of program, destroy context first,then free bm_handle
  delete p_ctx;
  bm_dev_free(bm_handle);
}
```

load_bmodel

```
bm_status_t load_bmodel(const void *bmodel_data, size_t size);
bm_status_t load_bmodel(const char *bmodel_file);
```

Load bmodel.

Bmodel can take the form of memory or files. It can be called by multiple threads. Return BM_SUCCESS in the case of successful loading; otherwise, return other error codes.

Multiple models can be continuously loaded but there cannot be repeated network names; otherwise, the loading will fail.

The use reference is shown as follows:

```
bm_status_t status;
status = p_ctx->load_bmodel(p_net1, net1_size); // p_net1 points to the bmodel memory buffer
assert(status == BM_SUCCESS);
status = p_ctx->load_bmodel("net2.bmodel"); // Specify the file route for the loaded bmodel
assert(status == BM_SUCCESS);
```

get_network_number

```
int get_network_number() const;
```

Get the number of networks loaded.

Each bmodel contains one to multiple networks. Loading bmodel each time will increase the number of networks.

get_network_names

```
void get_network_names(std::vector<const char *> *names) const;
```

Get the name of network loaded and save it to names. Note: This input vector will be clear firstly, then push_back all names of all networks.

The use reference is shown as follows:

```
std::vector<const char *> net_names;
p_ctx->get_network_names(&net_names);
for(auto name : net_names) {
    std::cout << name << std::endl;
}
```

get_network_info

```
const bm_net_info_t *get_network_info(const char *net_name) const;
```

Get the information of a specific network through the network name.

If net_name is available, return the network information structure pointer of bm_net_info_t, including the number, names and types of its inputs and outputs. For details, refer to the bm_net_info_t structure. If net_name is not available, return Null.

The use reference is shown as follows:

```
auto net1_info = p_ctx->get_network_info("net1");
if (net1_info == NULL) {
   std::cout << "net1 is not exist";
} else {
   std::cout << "net1 input num: " << net1_info->input_num;
}
```

handle

```
bm_handle_t handle() const;
```

Get the device handle of context, which is the same with the bm_handle loaded by the constructor. Handle is used when bm_xxxx class interfaces are called.

trace

```
void trace() const;
```

This interface is used for debugging. It can check context data and print some information about context to faciliate debugging.

class Network

Class Network is used to infer a specific network, which is selected among the loaded Context networks. This class can automatically request the device memory of inputs and outputs for the network. You may set the tensor of input and output if you need your device memory.

Constructor and destructor

```
Network(const Context &ctx, const char *net_name, int stage_id = -1);
virtual ~Network();
```

Constructor and destructure of network.

ctx is the Context instance mentioned above. Net_name means the name of the network installed in ctx and is used to create a network instance.

Stage_id refers to the sub-serial No. of the stage for a network. If stage_id is equal to -1, it indicates the user intends to reshape the shapes of all input tensors. For the sub-serial No. of a specific stage, the input tensors of network is fixed as the shape of this stage and cannot be reshaped later.

The use reference is shown as follows:

```
//net1, the shapes of input tensors can be reshaped
Network net1(*p_ctx, "net1");
//net2, the shape of stage [1] in bm_net_info_ is adopted and will not be reshaped later.
Network net2(*p_ctx, "net2", 1);
```

Inputs

```
const std::vector<Tensor *> &Inputs();
```

Get all input tensors.

Before inferring this network, get its input tensors first and then set all input sensors. For example, set theirs shape and data or device mem.

The use reference is shown as follows:

```
// Initialize the inputs of net1, supposing it has two inputs
auto &inputs = net1.Inputs();
inputs[0]->Reshape(shape0);
inputs[1]->Reshape(shape1);
// device_mem0 and device_mem1 have the data to be input
inputs[0]->set_device_mem(device_mem0);
inputs[1]->set_device_mem(device_mem1);

// Initialize the inputs of net2, supposing it has one input
auto &inputs = net2.Inputs();
// inputs[0]->Reshape(shape0); //  error, cannot be modified
// Suppose the data to be input is in the system memory and data0 is the data pointer.
inputs[0]->CopyFrom(data0);
```

Input

```
Tensor *Input(const char *tensor_name);
```

Get input tensor through input name.

Forward

```
bm_status_t Forward(bool sync = true) const;
```

Network inference.

Call Forward for inference upon the data of inputs are ready.

When sync is true, the interface will wait for the completion of inference. The interface is asynchronous and it is under the way and does not necessarily end when the interface exits. In this case, it is necessary to call the bm_thread_sync interface to ensure the completion of its inference.

Special note: The entire inference process occurs on device memory. So, the input data must have stored in the device mem of input tensors before inference. Upon the ending of inference, the result data is also saved to the device mem of output tensors.

The use reference is shown as follows:

```
// net1 inference
net1.Forward();
// et2 inference
net2.Forward(false);
bm_thread_sync(p_ctx->hand());
```

Outputs

```
const std::vector<Tensor *> &Outputs();
```

Get output tensors.

Before forward inference, you can change the device_mem in output tensors so that the inference result is saved to the device mem you specify or change the store mode of output sensors so that the inference result is saved to the store mode specified.

Only upon the completion of Forward inference, the shape of output tensors and data in device_mem are valid.

Output

```
Tensor *Output(const char *tensor_name);
```

Get output sensors through output names.

info

```
const bm_net_info_t *info() const;
```

Get the information of the network.

class Tensor

It is used for managing input sensors and output sensors of the network. You can not create Tensor on your own as Tensors are automatically created when the Network class is generated. So, both the constructor and the destructor are not public.

CopyTo

```
bm_status_t CopyTo(void *data) const;
bm_status_t CopyTo(void *data, size_t size, uint64_t offset = 0) const;
```

Copy the data of device mem for tensor to the system memory.

Data is a pointer that points at system memory data; size is used to specify the size of the copy and offset is used to specify the offset. When size and offset are not specified, copy the data of the entire tensor or the size of ByteSize ().

If the user needs to copy the output result to the system memory, after the inference ends, it is necessary to call CopyTo to copy the data to the system memory.

CopyFrom

```
bm_status_t CopyFrom(const void *data);
bm_status_t CopyFrom(const void *data, size_t size, uint64_t offset = 0);
```

Copy the system memory data to the device mem of tensor.

Data is a pointer that points at system memory data; size is used to specify the size of the copy and offset is used to specify the offset. When size and offset are not specified, copy the data of the entire tensor, that is the size of ByteSize ().

You should call CopyFrom and copy the data to the system memory before inference if you need to copy the data to the corresponding input tensor.

Reshape

```
bm_status_t Reshape(const bm_shape_t &shape);
```

Set the shape of tensor.

It is mainly used to change the shape of the input sensor and is meaningless for the output tensor as its shape is obtained through inference.

ByteSize

```
size_t ByteSize() const;
```

Get the size of tensor data, in byte, which is calculated by multiplying the numbe of elements and the number of bytes for element types.

num_elements

```
uint64_t num_elements() const;
```

Get the number of tensor elements. It is calculated by using the formula dims[0] * dims[1] * …* dims[num_dims-1]. Return 1 when num_dims is 0 (scalar element).

tensor

```
const bm_tensor_t *tensor() const;
```

Get the bm_tensor_t structure of tensor, which includes the shape, data type, device mem and store mode of tensor.

set_store_mode

```
void set_store_mode(bm_store_mode_t mode) const;
```

Set the store mode of tensor.

Before inference, you may configure the store mode of the input to specify the store mode of the input data or configure the store mode of output to indicate the data storage mode upon inference. In the absence of configuration, it is BM_STORE_1N by default.

set_device_mem

```
bm_status_t set_device_mem(const bm_device_mem_t &device_mem);
```

Set the device mem of the tensor.

Before inference, you can configure the device mem of the input to specify the store position of the input data or configure the device mem of output to indicate the store position of output.

Both input and output will be stored in the device mem automatically requested by network if you set nothing.

Additionally, you can configure the size of device mem, which cannot be smaller than ByteSize (); otherwise, errors will be returned due to the failure in storing the data of the entire tensor.

Global APIs

ByteSize

```
size_t ByteSize(bm_data_type_t type);                 // byte size of data type
```

Get the byte size of the data type, for example, the byte size of BM_FLOAT32 is 4 and that of BM_INT8 is 1.

```
size_t ByteSize(const bm_device_mem_t &device_mem);  // byte size of device memory
```

Get the byte size of device mem, that is the size of its storage space.

```
size_t ByteSize(const bm_tensor_t &tensor);           // byte size of origin tensor
```

Get the byte size of bm_tensor_t, which is equal to (number of tensor elements)*(byte size of the data type of tensor).

```
size_t ByteSize(const Tensor &tensor);                // byte size of tensor
```

Get the byte size of Tensor, which is equal to (number of tensor elements)* (byte size of the data type of tensor), which is identical with tensor.ByteSize()

Count

```
/*
dims[0] * dims[1] * dims[...] * dims[num_dims-1]
*/
uint64_t Count(const bm_shape_t &shape);
uint64_t Count(const bm_tensor_t &tensor);
uint64_t Count(const Tensor &tensor);
```

Get the number of elements or the product of the number of individual dimensions. Return 1 if num_dims is 0.

IsSameShape

```
/*
compare whether shape dims is the same
*/
bool IsSameShape(const bm_shape_t &left, const bm_shape_t &right);
```

Compare if both shapes are the same. Return true if yes, otherwise return false.

The interface is considered to be of the same shape only if num_dims and the corresponding dims[0], dims[1], ⋯dims[num_dims-1] are the same.

3.1.4 Multi-thread Program

Both C and C++ interfaces for the runtime aforementioned are thread safe, but other interfaces reserved for compatibility with the old version are not necessarily thread safe, so they are not recommended.

There are two types of common use methods:

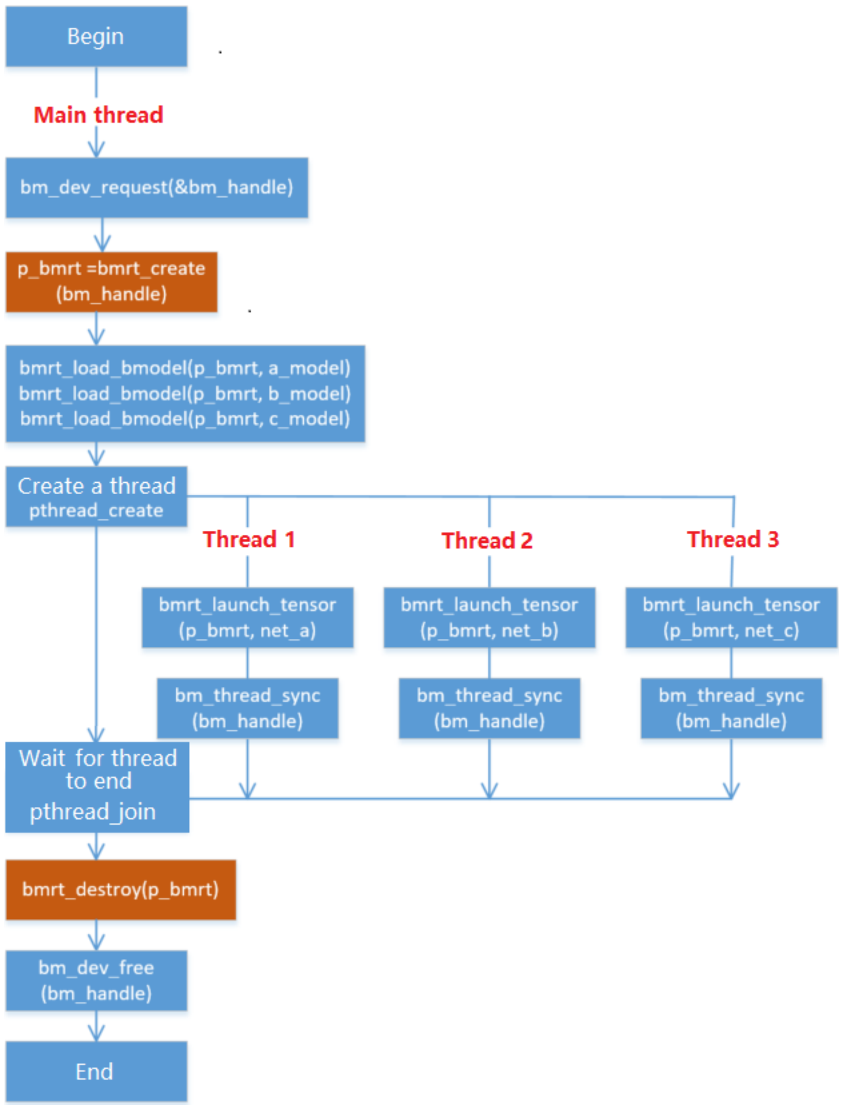
- Create a bmruntime and carry out multi-thread inference of networks after loading all models.
- One bmruntime is created for each thread. Load the model required by the thread for network inference.

single runtime

Single runtime can be used to load several different models. It should be noted that there can not be the same network between multiple models; otherwise, they will be considered to be in conflict. Similarly, the same model can only be loaded once.

Multi-thread inference is made on the loaded network through this runtime. The networks in multiple threads may be the same or different.

The programming model is shown as follows:


Begin
Main thread
bm_dev_request(&bm_handle)
p_bmrt =bmrt_create (bm_handle)
bmrt_load_bmodel(p_bmrt, a_model)
bmrt_load_bmodel(p_bmrt, b_model)
bmrt_load_bmodel(p_bmrt, c_model)
Create a thread pthread_create
Thread 1
Thread 2
Thread 3
bmrt_launch_tensor (p_bmrt, net_a)
bmrt_launch_tensor (p_bmrt, net_b)
bmrt_launch_tensor (p_bmrt, net_c)
bm_thread_sync (bm_handle)
bm_thread_sync (bm_handle)
bm_thread_sync (bm_handle)
Wait for thread to end pthread_join
bmrt_destroy(p_bmrt)
bm_dev_free (bm_handle)
End

The figure uses the C interface as an example.

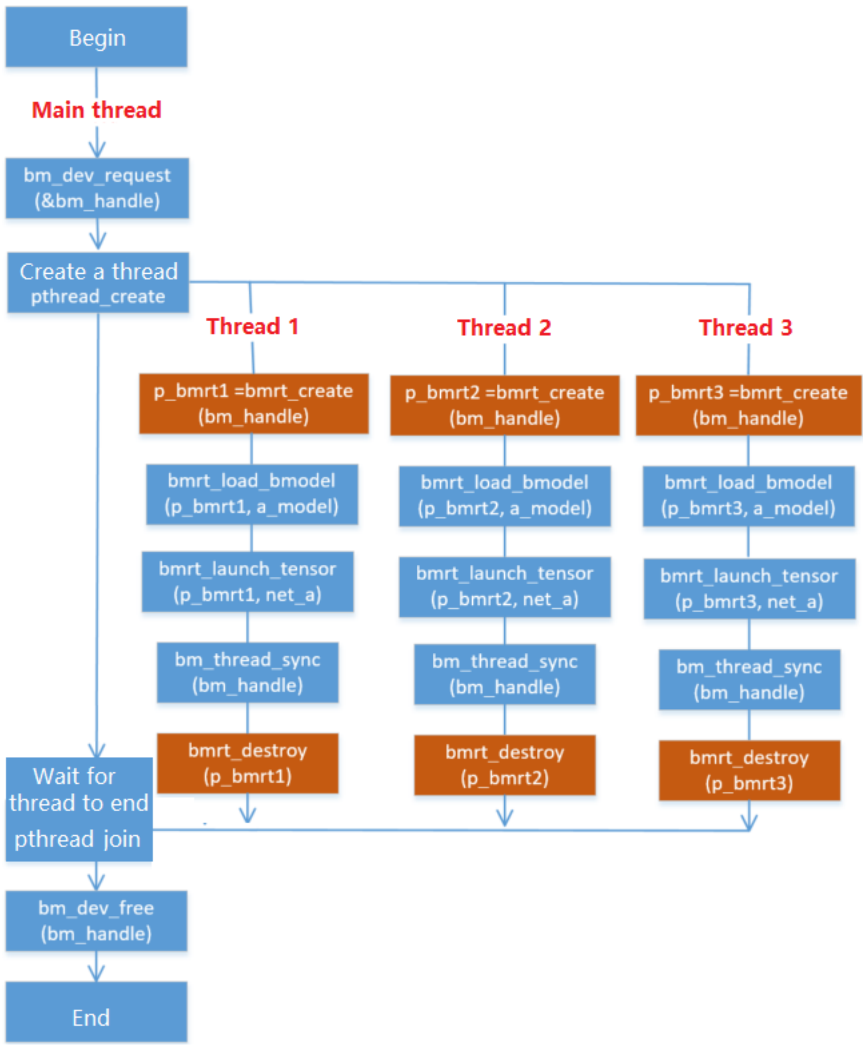
For the C++ interface, create a single Context instance and then load the network model via load_bmodel.

Next, create network instances in multiple threads for inference. The networks for the same instance may be the same or different.

multi runtime

You can create multiple threads, each creating a bumruntime. The loading model of each bmruntime is independent, with the same model loaded among them.

The programming model is shown as follows:


Begin
Main thread
bm_dev_request (&bm_handle)
Create a thread pthread_create
Thread 1
Thread 2
Thread 3
p_bmrt1 =bmrt_create (bm_handle)
p_bmrt2 =bmrt_create (bm_handle)
p_bmrt3 =bmrt_create (bm_handle)
bmrt_load_bmodel (p_bmrt1, a_model)
bmrt_load_bmodel (p_bmrt2, a_model)
bmrt_load_bmodel (p_bmrt3, a_model)
bmrt_launch_tensor (p_bmrt1, net_a)
bmrt_launch_tensor (p_bmrt2, net_a)
bmrt_launch_tensor (p_bmrt3, net_a)
bm_thread_sync (bm_handle)
bm_thread_sync (bm_handle)
bm_thread_sync (bm_handle)
bmrt_destroy (p_bmrt1)
bmrt_destroy (p_bmrt2)
bmrt_destroy (p_bmrt3)
Wait for thread to end pthread join
bm_dev_free (bm_handle)
End

The figure shows an example of the C interface.

For the C++ interface, create a single Context instance and then load the network model via load_bmodel.

how to choose

The difference between the two types of multi-thread programming methods is as follows:

A single runtime, with each network has only one neuron memory

So, using a single runtime means small neuron memory consumption. However, you need to wait for the neuron space of the network to be free if you perform multi-thread inference on the same network.

When there are multiple runtimes, each loads the same network. There is no need to wait for the neuron space to be free when loading the same network, but a lot of neuron memory will be consumed.

The option can be selected based on the following criteria and according to the user’s business conditions.

Please use multiple runtimes if it is necessary to perform multiple-thread inference on the same network; otherwise, use a single runtime.

CHAPTER 4

BMRuntime Code Examples

4.1 BMRuntimeCode Examples

4.1.1 Example with basic C interface

This section introduces the use of the runtime C interface through examples. One is common examples, which can be used on PCIE or SoC; the other uses mmap, which can only be used on SoC.

Common Example

Example description:

- Create bm_handle and a runtime instance.
- Load bmodel, which has one testnet network, two inputs and two outputs.
- Prepare input tensors, including the shape and data of each input.
- Start inference.
- Upon the ending of inference, copy the result data in output_tensor to the system memory.
- Before exiting from the program, release device mem, runtime instances and bm_handle.

It should be noted that in the example, the output data is copied to the system memory through the bm_memcpy_s2d_partial interface rather than the bm_memcpy_s2d interface. This is because the former specifies the size of the tensor, and the latter copies the whole according to the size of the device mem. The actual size of the output tensor is less than or equal to the size of the device mem, so using bm_memcp_s2d may overflow memory. For this reason, it is recommended to use the bm_memcpy_x2x_partial interface instead of the bm_memcpy_x2x interface.

```
#include "bmruntime_interface.h"

void bmrt_test() {
  // request bm_handle
  bm_handle_t bm_handle;
  bm_status_t status = bm_dev_request(&bm_handle, 0);
  assert(BM_SUCCESS == status);

  // create bmruntime
  void *p_bmrt = bmrt_create(bm_handle);
  assert(NULL != p_bmrt);

  // load bmodel by file
  bool ret = bmrt_load_bmodel(p_bmrt, "testnet.bmodel");
  assert(true == ret);

  auto net_info = bmrt_get_network_info(p_bmrt, "testnet");
  assert(NULL != net_info);

  // init input tensors
  bm_tensor_t input_tensors[2];
  status = bm_malloc_device_byte(bm_handle, &input_tensors[0].device_mem,
                                 net_info->max_input_bytes[0]);
  assert(BM_SUCCESS == status);
  input_tensors[0].dtype = BM_INT8;
  input_tensors[0].st_mode = BM_STORE_1N;
  status = bm_malloc_device_byte(bm_handle, &input_tensors[1].device_mem,
                                 net_info->max_input_bytes[1]);
  assert(BM_SUCCESS == status);
  input_tensors[1].dtype = BM_FLOAT32;
  input_tensors[1].st_mode = BM_STORE_1N;

  // init output tensors
  bm_tensor_t output_tensors[2];
  status = bm_malloc_device_byte(bm_handle, &output_tensors[0].device_mem,
                                 net_info->max_output_bytes[0]);
  assert(BM_SUCCESS == status);
  status = bm_malloc_device_byte(bm_handle, &output_tensors[1].device_mem,
                                 net_info->max_output_bytes[1]);
  assert(BM_SUCCESS == status);

  // before inference, set input shape and prepare input data
  // here input0/input1 is system buffer pointer.
  input_tensors[0].shape = {2, {1,2}};
  input_tensors[1].shape = {4, {4,3,28,28}};
  bm_memcpy_s2d_partial(bm_handle, input_tensors[0].device_mem, (void *)input0,
                        bmrt_tensor_bytesize(&input_tensors[0]));
  bm_memcpy_s2d_partial(bm_handle, input_tensors[1].device_mem, (void *)input1,
```

(continues on next page)

(continued from previous page)

```
                    bmrt_tensor_bytesize(&input_tensors[1]));

  ret = bmrt_launch_tensor_ex(p_bmrt, "testnet", input_tensors, 2,
                              output_tensors, 2, true, false);
  assert(true == ret);

  // sync, wait for finishing inference
  bm_thread_sync(bm_handle);

  /**************************************************************/
  // here all output info stored in output_tensors, such as data type, shape, device_mem.
  // you can copy data to system memory, like this.
  // here output0/output1 is system buffers to store result.
  bm_memcpy_d2s_partial(bm_handle, output0, output_tensors[0].device_mem,
                        bmrt_tensor_bytesize(&output_tensors[0]));
  bm_memcpy_d2s_partial(bm_handle, output1, output_tensors[1].device_mem,
                        bmrt_tensor_bytesize(&output_tensors[1]));
  ......     // do other things
  /**************************************************************/

  // at last, free device memory
  for (int i = 0; i < net_info->input_num; ++i) {
    bm_free_device(bm_handle, input_tensors[i].device_mem);
  }
  for (int i = 0; i < net_info->output_num; ++i) {
    bm_free_device(bm_handle, output_tensors[i].device_mem);
  }

  bmrt_destroy(p_bmrt);
  bm_dev_free(bm_handle);
}
```

MMAP Example

The function of this example is the same as the previous one, but it uses mmap to map it to the application for direct access rather than copying device mem data. Its efficiency is higher than the that of above example, but it can only be used under SoC.

```
#include "bmruntime_interface.h"

void bmrt_test() {
  // request bm_handle
  bm_handle_t bm_handle;
  bm_status_t status = bm_dev_request(&bm_handle, 0);
  assert(BM_SUCCESS == status);

  // create bmruntime
  void *p_bmrt = bmrt_create(bm_handle);
  assert(NULL != p_bmrt);
```

(continues on next page)

(continued from previous page)

```
// load bmodel by file
bool ret = bmrt_load_bmodel(p_bmrt, "testnet.bmodel");
assert(true == ret);

auto net_info = bmrt_get_network_info(p_bmrt, "testnet");
assert(NULL != net_info);

bm_tensor_t input_tensors[2];
bmrt_tensor(&input_tensors[0], p_bmrt, BM_INT8, {2, {1,2}});
bmrt_tensor(&input_tensors[1], p_bmrt, BM_FLOAT32, {4, {4,3,28,28}});

void *input[2];
status = bm_mem_mmap_device_mem(bm_handle, &input_tensors[0].device_mem,
                                (uint64_t*)&input[0]);
assert(BM_SUCCESS == status);
status = bm_mem_mmap_device_mem(bm_handle, &input_tensors[1].device_mem,
                                (uint64_t*)&input[1]);
assert(BM_SUCCESS == status);

// write input data to input[0], input[1]
......

// flush it
status = bm_mem_flush_device_mem(bm_handle, &input_tensors[0].device_mem);
assert(BM_SUCCESS == status);
status = bm_mem_flush_device_mem(bm_handle, &input_tensors[1].device_mem);
assert(BM_SUCCESS == status);

// prepare output tensor, and launch
assert(net_info->output_num == 2);

bm_tensor_t output_tensors[2];
ret = bmrt_launch_tensor(p_bmrt, "testnet", input_tensors,2,
                         output_tensors, 2);
assert(true == ret);

// sync, wait for finishing inference
bm_thread_sync(bm_handle);

/**************************************************************/
// here all output info stored in output_tensors, such as data type, shape, device_mem.
// you can access system memory, like this.
void * output[2];
status = bm_mem_mmap_device_mem(bm_handle, &output_tensors[0].device_mem,
                                (uint64_t*)&output[0]);
assert(BM_SUCCESS == status);
status = bm_mem_mmap_device_mem(bm_handle, &output_tensors[1].device_mem,
                                (uint64_t*)&output[1]);
assert(BM_SUCCESS == status);
status = bm_mem_invalidate_device_mem(bm_handle, &output_tensors[0].device_mem);
```

(continues on next page)

(continued from previous page)

```
  assert(BM_SUCCESS == status);
  status = bm_mem_invalidate_device_mem(bm_handle, &output_tensors[1].device_mem);
  assert(BM_SUCCESS == status);
  // do other things
  // users can access output by output[0] and output[1]
  ......
  /****************************************************************/

  // at last, unmap and free device memory
  for (int i = 0; i < net_info->input_num; ++i) {
    status = bm_mem_unmap_device_mem(bm_handle, input[i],
                                  bm_mem_get_device_size(input_tensors[i].device_mem));
    assert(BM_SUCCESS == status);
    bm_free_device(bm_handle, input_tensors[i].device_mem);
  }
  for (int i = 0; i < net_info->output_num; ++i) {
    status = bm_mem_unmap_device_mem(bm_handle, output[i],
                                  bm_mem_get_device_size(output_tensors[i].device_mem));
    assert(BM_SUCCESS == status);
    bm_free_device(bm_handle, output_tensors[i].device_mem);
  }

  bmrt_destroy(p_bmrt);
  bm_dev_free(bm_handle);
}
```

4.1.2 Example with basic C++ interface

This section introduces the use of the runtime C++ interface through examples. One is common examples, which can be used on PCIE or SoC; the other uses mmap, which can only be used on SoC.

Common Example

Example description:

- Create bm_handle and a context instance.
- Load bmodel, which has one testnet network, two inputs and two outputs.
- Prepare input tensors, including the shape and data of each input.
- Start inference.
- Upon the ending of inference, copy the result data in output_tensors to the system memory.
- Before exiting from the program, release bm_handle.

Two Networks are instantiated for the Context' s testnet network to demonstrate the following points:

- When stage is not specified by Network, each input requires Reshape to set the input shape; when stage is specified by Network, configure input according to the shape of stage. There is no need for you to reshape.
- The same network name can be instantiated into multiple Networks without any influence between them. Similarly, each network can be inferred among multiple threads.

```
#include "bmruntime_cpp.h"

using namespace bmruntime;

void bmrt_test()
{
  // create Context
  Context ctx;

  // load bmodel by file
  bm_status_t status = ctx.load_bmodel("testnet.bmodel");
  assert(BM_SUCCESS == status);

  // create Network
  Network net1(ctx, "testnet"); // may use any stage
  Network net2(ctx, "testnet", 0); // use stage[0]

  /**************************************************************/
  // net1 example
  {
    // prepare input tensor, assume testnet has 2 input
    assert(net1.info()->input_num == 2);
    auto &inputs = net1.Inputs();
    inputs[0]->Reshape({2, {1, 2}});
    inputs[1]->Reshape({4, {4, 3, 28, 28}});
    // here input0/input1 is system buffer pointer to input datas
    inputs[0]->CopyFrom((void *)input0);
    inputs[1]->CopyFrom((void *)input1);

    // do inference
    status = net1.Forward();
    assert(BM_SUCCESS == status);

    // here all output info stored in output_tensors, such as data type, shape, device_mem.
    // you can copy data to system memory, like this.
    // here output0/output1 is system buffers to store result.
    auto &outputs = net1.Outputs();
    outputs[0]->CopyTo(output0);
    outputs[1]->CopyTo(output1);
    ......  // do other things
  }
```

(continues on next page)

(continued from previous page)

```
  /****************************************************************/
  // net2 example
  // prepare input tensor, assume testnet has 2 input
  {
    assert(net2.info()->input_num == 2);
    auto &inputs = net2.Inputs();
    inputs[0]->CopyFrom((void *)input0);
    inputs[1]->CopyFrom((void *)input1);
    status = net2.Forward();
    assert(BM_SUCCESS == status);
    // here all output info stored in output_tensors
    auto &outputs = net2.Outputs();
    ...... // do other things
  }
}
```

MMAP Example

This example only instantiates one network, mainly illustrating how to use mmap.

```
#include "bmruntime_cpp.h"

using namespace bmruntime;

void bmrt_test()
{
  // create Context
  Context ctx;

  // load bmodel by file
  bm_status_t status = ctx.load_bmodel("testnet.bmodel");
  assert(BM_SUCCESS == status);

  // create Network

  Network net(ctx, "testnet", 0); // use stage[0]

  // prepare input tensor, assume testnet has 2 input
  assert(net.info()->input_num == 2);
  auto &inputs = net.Inputs();

  void *input[2];
  bm_handle_t bm_handle = ctx.handle();
  status = bm_mem_mmap_device_mem(bm_handle, &(inputs[0]->tensor()->device_mem),
                                  (uint64_t*)&input[0]);
  assert(BM_SUCCESS == status);
  status = bm_mem_mmap_device_mem(bm_handle, &(inputs[1]->tensor()->device_mem),
                                  (uint64_t*)&input[1]);
  assert(BM_SUCCESS == status);
```

(continues on next page)

(continued from previous page)

```
  // write input data to input[0], input[1]
  ......

  // flush it
  status = bm_mem_flush_device_mem(bm_handle, &(inputs[0]->tensor()->device_mem));
  assert(BM_SUCCESS == status);
  status = bm_mem_flush_device_mem(bm_handle, &(inputs[1]->tensor()->device_mem));
  assert(BM_SUCCESS == status);

  status = net.Forward();
  assert(BM_SUCCESS == status);
  // here all output info stored in output_tensors
  auto &outputs = net.Outputs();

  // mmap output
  void * output[2];
  status = bm_mem_mmap_device_mem(bm_handle, &(outputs[0]->tensor()->device_mem),
                                  (uint64_t*)&output[0]);
  assert(BM_SUCCESS == status);
  status = bm_mem_mmap_device_mem(bm_handle, &(outputs[1]->tensor()->device_mem),
                                  (uint64_t*)&output[1]);
  assert(BM_SUCCESS == status);
  // invalidate it
  status = bm_mem_invalidate_device_mem(bm_handle, &(outputs[0]->tensor()->device_
↪mem));
  assert(BM_SUCCESS == status);
  status = bm_mem_invalidate_device_mem(bm_handle, &(outputs[1]->tensor()->device_
↪mem));
  assert(BM_SUCCESS == status);

  // user can access output by output[0] and output[1]
  ......

  // at last, unmap bm_handle
  status = bm_mem_unmap_device_mem(bm_handle, input[0],
                                  bm_mem_get_device_size(inputs[0]->tensor()->device_mem));
  assert(BM_SUCCESS == status);
  status = bm_mem_unmap_device_mem(bm_handle, input[1],
                                  bm_mem_get_device_size(inputs[1]->tensor()->device_mem));
  assert(BM_SUCCESS == status);
  status = bm_mem_unmap_device_mem(bm_handle, output[0],
                                  bm_mem_get_device_size(outputs[0]->tensor()->device_mem));
  assert(BM_SUCCESS == status);
  status = bm_mem_unmap_device_mem(bm_handle, output[1],
                                  bm_mem_get_device_size(outputs[1]->tensor()->device_mem));
  assert(BM_SUCCESS == status);
}
```

CHAPTER 5

bmrt_test Use and bmodel Verification

5.1 bmrt_test Use and bmodel Verification

5.1.1 bmrt_test tool

bmrt_test is a tool for testing the correctness and actual running performance of bmodel based on the bmruntime interface. It contains the following functions:

1. Directly inferring random data bmodel to verify the integrity and operability of bmodel;
2. Directly using bmodel for inference through fixed input data, comparing the output and reference data and verifying the correctness of the data;
3. Testing the actual running time of bmodel;
4. Profiling bmodel through the bmprofile mechanism.

5.1.2 bmrt_test Parameter Description

Table 5.1: bmrt_test main parameter description

args	type	Description
context_dir	string	The result folder compiled by the model: the comparison data is also generated during compilation and the comparison is enabled by default. When comparison is disabled, the folder contains a compilation.bmodel file. When the comparison is enabled, the folder should contain three files: compilation.bmodel, input_ref_data.dat, output_ref_data.dat
bmodel	string	Choose one from context_dir and bmodel, and specify the .bmodel file directly.Comparison is disabled by default.
devid	int	Optional, specifying the running device by id on multi-core platforms, with the default value being 0.
compare	bool	Optional, 0 indicates comparision is disabled and 1 indicates comparison is enabled.
accuracy_f	float	Optional, specifying the float data comparison error threshold, with the default value being 0.01.
accuracy_i	int	Optional, speciying the integer data comparison error threshold, with the default value being 0.
shapes	string	Optional, specifying the input shapes in testing, with the compilation input shape for bmodel being used by default. The format is “[x,x,x,x],[x,x]” , corresponding to the sequence and number of inputs for the model.
loopnum	int	Optional, specifying the times of continuous operations, with the default value being 1.
thread_num	int	Optional, specifying the number of running threads, with the default value being 1, and testing the correctness of multiple threads.
net_idx	int	Optional, selecting the net to run by the serial No. in a bmodel that contains multiple nets.
stage_idx	int	Optional, selecting the stage to run by the serial No. in a bmodel that contains multiple stages.
subnet_time	bool	Optional, indicating whether to display the subnet time of bmodel.

5.1.3 bmrt_test Output

```
[BMRT][bmrt_test:1250] INFO:net[resnet50-v2] stage[0], launch total time is 6996 us
↪(npu 6801 us, cpu 195 us), (launch func time 164 us, sync 6834 us)
[BMRT][bmrt_test:1257] INFO:+++ The network[resnet50-v2] stage[0] output_data
↪+++
[BMRT][print_array:766] INFO:output data #0 shape: [1 1000 ] < -0.437744 2.16406 -
↪3.0332 -2.36719 -1.19238 0.836426 -2.34766 -1.54004 2.42188 -0.641602 3.03516 0.
↪797852 1.31055 1.50879 -0.870605 1.2998 ... > len=1000
[BMRT][bmrt_test:1271] INFO:==>comparing output in mem #0 ...
[BMRT][bmrt_test:1304] INFO:+++ The network[resnet50-v2] stage[0] cmp success
↪+++
[BMRT][bmrt_test:1319] INFO:load input time(s): 0.008669
[BMRT][bmrt_test:1320] INFO:pre alloc  time(s): 0.000974
[BMRT][bmrt_test:1321] INFO:calculate  time(s): 0.006998
[BMRT][bmrt_test:1322] INFO:get output time(s): 0.000076
[BMRT][bmrt_test:1323] INFO:compare    time(s): 0.000845
```

Main focuses:

(1) Pure inference time of the model, excluding loading the input and getting the output.

(2) Inference data display: The information of the successful comparison equation will be displayed if the comparison is enabled.

(3) Use s2d to load the input data time, Usually, the pre-processing will put the data directly on the device without such time consumption.

(4) Use d2s to take out the output data time, which usually means the data transmission time on the pcie. Mmap, with a faster speed, can be used on the SOC.

5.1.4 Common Methods of bmrt_test

```
bmrt_test --context_dir bmodel_dir  # Run bmodel and compare the data.The
# bmodel_dir should include compilation.bmodel/input_ref_data.dat/output_ref_
↪data.dat.
bmrt_test --context_dir bmodel_dir  --compare=0 # Run bmodel, The bmodel_dir
# should include compilation.bmode.
bmrt_test --bmodel xxx.bmodel # Directly run bmodel without comparing data
bmrt_test --bmodel xxx.bmodel --stage_idx 0  --shapes "[1,3,224,224]" # Run the
# multi-stage bmodel model and specify the bmodel for running stage 0.

bmrt_test --bmodel xxx.bmodel --core_list 0,1
# run the bmodel on the deep learning processor core0 and core1 at the same time
# note that the bmodel is multi-core compiled and can be architected to support multi-
↪core
# the value in core_list is at least 0 and cannot be greater than the number of ( deep
↪learning processor cores - 1 ).
```

(continues on next page)

(continued from previous page)

```
# The following instructions are functions provided by using environmental variables
# and bmruntime and can be used by other applications.
BMRUNTIME_ENABLE_PROFILE=1 bmrt_test --bmodel xxx.bmodel # Generate
# profile data: bmprofile_data-x
BMRT_SAVE_IO_TENSORS=1 bmrt_test --bmodel xxx.bmodel  # Save the
# model inference data as input_ref_data.dat.bmrt and output_ref_data.dat.bmrt.
```

5.1.5 Comparison Data Generation and Verification Example

1. Upon the completion of model compilation, run with comparing the model.

   When compiling the model in deploy stage, you must indicate --test_input and --test_reference. input_ref_data.dat and output_ref_data.dat files will be generated in the compilation output folder.

   Then, execute ‘bmrt_test --context_dir bmodel_dir’ to verify the correctness of the model inference data.

2. Comparison of pytorch original model and compiled bmodel data

   Convert the input input_data and output output_data of the pytorch model to numpy array (torch tensor can use tensor.numpy()), and then save the file (see the codes below).

```
# Single inputs and single outputs
input_data.astype(np.float32).tofile("input_ref_data.dat")  # astype will
# convert according to the input data type of bmodel
output_data.astype(np.float32).tofile("output_ref_data.dat")  # astype will
# convert according to the output data type of bmodel

# Multiple inputs and multiple outputs
with open("input_ref_data.dat", "wb") as f:
    for input_data in input_data_list:
        f.write(input_data.astype(np.float32).tobytes())  # astype will convert
        # according to the input data type of bmodel
with open("output_ref_data.dat", "wb") as f:
    for output_data in output_data_list:
        f.write(output_data.astype(np.float32).tobytes())  # astype will convert
        # according to the output data type of bmodel
```

   Put the generated input_ref_data.dat and output_ref_data.dat in the bmodel_dir file folder and then in ‘bmrt_test --context_dir bmodel_dir’ to see if the result is a comparison error.

5.1.6 FAQs

1. Will data comparison error occur when compiling the model?

Our bmcompiler internally uses 0.01 as the comparison threshold, which may exceed the range and report an error in a few cases.

If there is any problem with the implementation on a certain layer, there will be a piece-by-piece comparison error, and we need to give feedback to our developers.

If there are sporadic errors in random positions, it may be caused by errors in the calculation of individual values. The reason is that random data is used when compiling, which cannot be ruled out. Therefore, it is recommended to add --cmp 0 when compiling, and verify whether the result is correct on the actual business program.

Another possibility is that there are random operators (such as uniform_random) or sorting operators (such as topk, nms, argmin, etc.) in the network, as the floating-point mantissa error of the input data will be generated in the previous calculation process, even if it is small, and will cause the difference in the indexes of sorted results. In this case, it can be seen that there is a difference in the order of the data with errors in the comparison, and it can only be tested in the actual business.