# BMRuntime 开发参考手册

SOPHGO

2025 年 10 月 27 日

# Contents

CHAPTER 1

# 声明

## 1.1 声明


**法律声明**



**注意**

您购买的产品、服务或特性等应受算能商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，算能对本文档内容不做任何明示或默示的声明或保证。由于产品版本升级或其他原因，本文档内容会不定期

进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

### 技术支持

**地址**
北京市海淀区丰豪东路 9 号院中关村集成电路设计园（ICPARK）1 号楼

**邮编**
100094

**网址**
https://www.sophgo.com/

**邮箱**
sales@sophgo.com

**电话**
+86-10-57590723 +86-10-57590724

### SDK 发布记录

| 版本 | 发布日期 | 说明 |
|---|---|---|
| V2.0.0 | 2019.09.20 | 第一次发布。 |
| V2.0.1 | 2019.11.16 | V2.0.1 版本发布。 |
| V2.0.3 | 2020.05.07 | V2.0.3 版本发布。 |
| V2.2.0 | 2020.10.12 | V2.2.0 版本发布。 |
| V2.3.0 | 2021.01.11 | V2.3.0 版本发布。 |
| V2.3.1 | 2021.03.09 | V2.3.1 版本发布。 |
| V2.3.2 | 2021.04.01 | V2.3.2 版本发布。 |
| V2.4.0 | 2021.05.23 | V2.4.0 版本发布。 |
| V2.5.0 | 2021.09.02 | V2.5.0 版本发布。 |
| V2.6.0 | 2021.01.30 | V2.6.0 版本修正后发布。 |
| V2.7.0 | 2022.03.16 | V2.7.0 版本发布, 20220531 发布补丁版本。 |
| V3.0.0 | 2022.07.16 | V3.0.0 版本发布。 |

CHAPTER 2

# BModel

## 2.1 BModel

### 2.1.1 BModel 介绍

bmodel 是面向算能深度学习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 tpu-mlir 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。

多 stage bmodel 说明：

> bmodel 里 stage 是用多种 input_shape 分别编译出 bmodel，然后用 tpu_model 将多个 bmodel 合并到一起成为一个 bmodel，而里面包含的每个 bmodel 是一个 stage。stage_num 就是指合并的 bmodel 个数。未经过合并的 bmodel 的 stage_num=1。当以某种 shape 运行模型时，bmruntime 会自动选择有相同输入 shape 的 bmodel 运行。
>
> 通过选择几种常用的输入 shape 合并，可以提升运行效率，并实现动态运行的效果比如分别以 [1,3,200,200]，[2,3,200,200] 的输入编译出两个 bmodel，合并后运行如果以 [2,3,200,200] 输入运行，则会自动找 [2,3,200,200] 的那个 bmodel 运行
>
> 也可以分别以 [1,3,200,200]，[1,3,100,100] 的输入编译出两个 bmodel，达到支持 200x200 和 100x100 输入的模型

静态 bmodel 说明：

1. 静态 bmodel 保存的是芯片上可直接使用的固定参数原子操作指令，深度学习处理器可以自动读取该原子操作指令，流水执行，中间无中断。
2. 静态 bmodel 被执行时，模型输入大小必须和编译时的大小相同。

3. 由于静态接口简单稳定，在新的 sdk 下编译出来的模型通常能在旧机器上运行，不用更新 firmware 刷机。需要注意的是，有些模型虽然指定的是静态编译，但有些算子必需有深度学习处理器内部 mcu 参与或主机处理器参与，如排序、nms、where、detect_out 这类逻辑运算比较多的算子，该部分会被切分成子网，用动态方式实现。如果更新 sdk 重新编译的这类部分是动态的模型，最好刷机或更新 firmware，以保证 sdk 和 runtime 是一致的。(可以通过 tpu_model --info xx.bmodel 的输出来判断，如果是 static 且 subnet number 为 1 时，是纯静态网络，具体可见 tpu_model 使用章节)。
4. 如果输入的 shape 只有固定离散的几种情况，可以使用上面说的多 stage bmodel 来达到动态模型的效果。

动态 bmodel 说明:

1. 动态 bmodel 保存的是每个算子的参数信息，并不能直接在深度学习处理器上运行。需要深度学习处理器内部的 mcu 逐层解析参数，进行 shape 及 dtype 推理，调用原子操作来实现具体的功能，故运行效率比静态 bmodel 稍差。
2. 在 bm168x 平台上运行时，最好打开 icache，否则运行比较慢。
3. 编译时，通过 shapes 参数来指定支持的最大 shape。在实际运行时，除了 c 维，其他维均支持可变。通常在可变 shape 情况过多，考虑用动态编译，否则推荐用多 stage bmodel 模式。
4. 动态 bmodel 为了保证参数可扩展以及兼容，会保证新 sdk 的 runtime 能运行旧版本 sdk 编译出的动态 bmodel。通常建议换新 sdk 后刷机，保证两者版本一致。

### 2.1.2 tpu_model 使用

通过 tpu_model 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。

目前支持以下六种使用方法:

1. 查看简要信息 (比较常用)

   tpu_model --info xxx.bmodel

   输出信息如下

```
bmodel version: B.2.2                    # bmodel的格式版本号
chip: BM1684                             # 支持的芯片类型
create time: Mon Apr 11 13:37:45 2024        # 创建时间

=============================================    #F
↪网络分割线，如果有多个net，会有多条分割线
net 0: [informer_frozen_graph]  static       # 网络名称为informer_frozen_graph，F
↪为static类型网络（即静态网络），如果是dynamic，为动态编译网络
------------                             #F
↪stage分割线，如果每个网络有多个stage，会有多个分割线
stage 0, core_num: x                      #F
↪第一个stage信息和对应的深度学习处理器core数量
subnet number: 41                         #F
```

(续下页)

(接上页)

```
↪该stage中子网个数，这个是编译时切分的，以支持在不同设备切换运行。通常子网个数
                                    # 越少越好
input: x_1, [1, 600, 9], float32, scale: 1    # 输入输出信息：名称、形状、量化的scale值
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:F
↪788755552) # 该模型在深度学习处理器上内存占用情况（以byte为单位），格式为：F
↪总占用内存大小（常量内存大小，指令内存大小，运行时数据内存占用大小）
host mem size: 8492192 (coeff: 32, runtime: 8492160)  #F
↪宿主机上内存占用情况（以byte为单位），格式为：F
↪总占用内存大小（常量内存大小，运行时数据内存大小）
```

2. 查看详细参数信息

   tpu_model --print xxx.bmodel

3. 分解

   tpu_model --extract xxx.bmodel

   将一个包含多个网络多种 stage 的 bmodel 分解成只包含一个网络的一个 stage 的各个 bmodel，分解出来的 bmodel 按照 net 序号和 stage 序号，命名为 bm_net0_stage0.bmodel、bm_net1_stage0.bmodel 等等。

4. 合并

   tpu_model --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。

   多个 bmodel 合并后：

   - 不同 net_name 的 bmodel 合并，接口会根据 net_name 选择对应的网络进行推理
   - 相同 net_name 的 bmodel 合并，会使该 net_name 的网络可以支持多种 stage(也就是支持不同的 input shape)。接口会根据用户输入的 shape，在该网络的多个 stage 中选择。对于静态网络，它会选择 shape 完全匹配的 stage；对于动态网络，它会选择最靠近的 stage。

   限制：同一个网络 net_name，使用 combine 时，要求都是静态编译，或者都是动态编译。暂时不支持相同 net_name 的静态编译和动态编译的 combine。

5. 合并文件夹

   tpu_model --combine_dir a_dir b_dir c_dir -o abc_dir

   同 combine 功能类似，不同的是，该功能除了合并 bmodel 外，还会合并用于测试的输入输出文件。它以文件夹为单位合并，文件夹中必须包含经过编译器生成的三个文件：input_ref_data.dat, output_ref_data.dat, compilation.bmodel。

6. 导出二进制数据

   tpu_model --dump xxx.bmodel start_offset byte_size out_file

   将 bmodel 中的二进制数据保存到一个文件中。通过 print 功能可以查看所有二进制数据的 [start, size]，对应此处的 start_offset 和 byte_size。

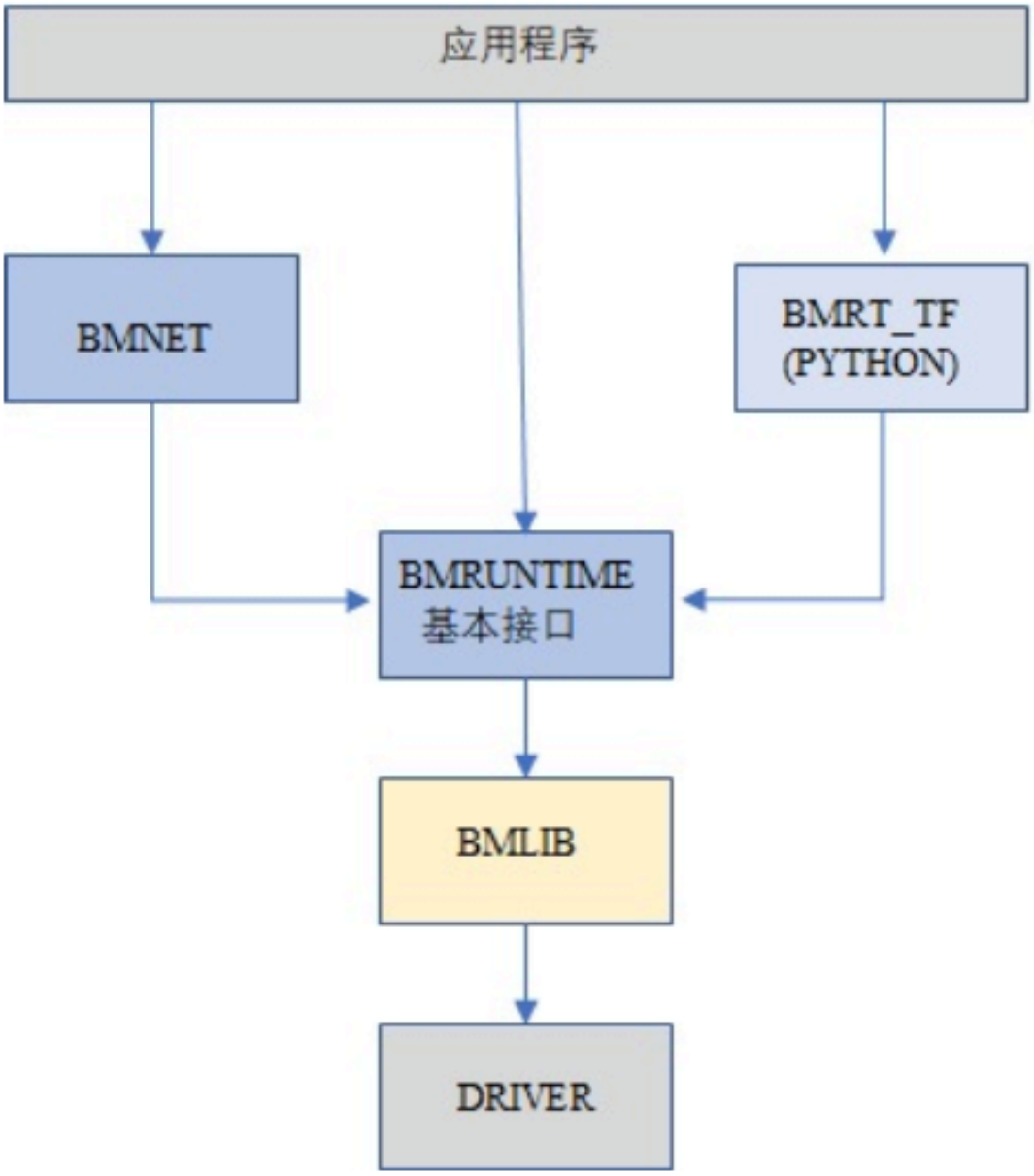
 

CHAPTER 3

# BMRuntime

## 3.1 BMRuntime

BMRuntime 用于读取 BMCompiler 的编译输出 (.bmodel)，驱动其在深度学习处理器中执行。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:




BMRuntime 有 C 和 C++ 两种接口；另外为了兼容上一代应用程序，保留一些接口，但不推荐新的应用程序继续使用。

本章节中的接口默认都是同步接口，有个别是异步接口 (由深度学习处理器执行功能，主机处理器可以继续往下执行)，会特别说明。

本章节分 4 个部分:

- BMLIB 接口：用于设备管理，不属于 BMRuntime，但需要配合使用，所以先介绍
- C 接口：BMRuntime 的 C 语言接口
- C++ 接口：BMRuntime 的 C++ 语言接口

- 多线程编程：介绍如何用 C 接口或者 C++ 接口进行多线程编程

### 3.1.1 BMLIB Interface

BMLIB 接口是 C 语言接口，对应的头文件是 bmlib_runtime.h，对应的 lib 库为 libbmlib.so

BMLIB 接口用于设备管理，包括设备内存的管理。

BMLIB 的接口很多，这里介绍应用程序通常需要用到的接口。

#### Device

##### bm_dev_request

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

用于请求一个设备，得到设备句柄 handle。其他设备接口 (bm_xxxx 类接口)，都需要指定这个设备句柄。

其中 devid 表示设备号，在 PCIE 模式下，存在多个设备时可以用于选择对应的设备；在 SoC 模式下，请指定为 0。

当请求成功时，返回 BM_SUCCESS；否则返回其他错误码。

##### bm_dev_free

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

用于释放一个设备。通常应用程序开始需要请求一个设备，退出前释放这个设备。

使用参考如下:

```
// 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);
```

申请指定大小的 device mem，size 为 device mem 的字节大小。

当申请成功时，返回 BM_SUCCESS；否则返回其他错误码。

#### bm_free_device

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

释放 device mem。任何申请的 device mem，不再使用的时候都需要释放。

使用参考如下：

```
// 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);
```

得到 device mem 的大小，以字节为单位。

### bm_memcpy_s2d

将在系统内存上的数据拷贝到 device mem。系统内存由 void 指针指定，device mem 由 bm_device_mem_t 类型指定。

拷贝成功，返回 BM_SUCCESS；否则返回其他错误码。

根据拷贝的大小和偏移，有以下三种：

```
// 拷贝的大小是device mem的大小，从src开始拷贝
/* [in]  handle
 * [out] dst
 * [in]  src
 */
bm_status_t bm_memcpy_s2d(bm_handle_t handle, bm_device_mem_t dst, void *src);
```

```
// size指定拷贝的字节大小，从src的offset偏移开始拷贝
/* [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指定拷贝的字节大小，从src开始拷贝
/* [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

将 device mem 中的数据拷贝到系统内存；拷贝成功，返回 BM_SUCCESS；否则返回其他错误码。

系统内存由 void 指针指定，device mem 由 bm_device_mem_t 类型指定。

根据拷贝的大小和偏移，有以下三种：

```
// 拷贝的大小是device mem的大小，从device mem的0偏移开始拷贝
/* [in]  handle
 * [out] dst
 * [in]  src
```

(续下页)

(接上页)

```
 */
bm_status_t bm_memcpy_d2s(bm_handle_t handle, void *dst, bm_device_mem_t src);
```

```
// size指定拷贝的字节大小，从device mem的offset偏移开始拷贝
/* [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指定拷贝的字节大小，从device mem的0偏移位置开始拷贝
/* [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);
```

将数据从一个 device mem 拷贝到另一个 device mem。

dst_offset 指定目标的偏移，src_offset 指定源的偏移，len 指定拷贝的大小。

**特别注意**: len 是以 dword 为单位，比如要拷贝 1024 个字节，则 len 需要指定为 1024/4=256。

### Device Memory Mmap

此小节介绍的接口，只有在 SoC 上有效。在 SoC 上，系统内存和 Device Memory 虽然是隔开的，但其实都是 DDR 上的内存。

可以通过 mmap，得到 Device Memory 的虚拟地址，从而可以被应用程序直接访问。

**特别注意**: NPU 访问 Device Memory，是直接访问 DDR，没有经过 cache；而应用程序访问的时候是经过 cache 的。

因此需要处理 cache 的一致性，也就是说：

- 应用程序通过虚拟地址修改了 Device Memory 的数据，在进行 NPU 推理前需要 flush，确保 cache 数据已经同步到了 DDR
- NPU 推理结束后修改了 Device Memory 的数据，应用程序通过虚拟地址访问前需要先 invalidate，确保 DDR 数据已经同步到了 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);
```

将 device mem 映射出来，得到虚拟地址。

成功返回 BM_SUCCESS；否则返回其他错误码。

#### 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);
```

映射的虚拟地址不再使用的时候需要 unmap，size 为 device mem 的大小，这个大小可以通过 bm_mem_get_device_size 得到。

### 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);
```

使 cache 失效，也就是确保 DDR 数据同步到了 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);
```

指定 device mem 的偏移和大小的范围内 cache 失效

### 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);
```

刷新 cache 数据，也就是确保 cache 数据同步到了 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);
```

指定 device mem 的偏移和大小的范围内 cache 刷新

### example

这里举例说明 mmap 接口的用法:

```
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);
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);
```

同步接口。通常 npu 推理是异步进行的，用户的 cpu 程序可以继续执行。本接口用于 cpu 程序中确保 npu 推理完成。本章介绍的接口没有特别说明，都是同步接口。只有个别异步接口，需要调用 bm_thread_sync 进行同步。

### 3.1.2 C Interface

BMRuntime 的 C 语言接口，对应的头文件为 bmruntime_interface.h，对应的 lib 库为 libbmrt.so。

用户程序使用 C 接口时建议使用该接口，该接口支持多种 shape 的静态编译网络，支持动态编译网络。

#### Tensor information

Tensor（张量）表示多维的数据，BMRuntime 中操作的数据为 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 用于表示数据类型。

#### 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 表示数据的存储方式。用户可以只关注 BM_STORE_1N 即可，若要关注底层并优化性能，此时才需要去关心 BM_STORE_2N 和 BM_STORE_4N。

BM_STORE_1N 是默认存储方式，用于数据类型，表示数据按正常方式存储。

BM_STORE_2N 只用于 BM_FLOAT16/BM_INT16/BM_UINT16，表示一个 32bit 的数据空间将放置 2 个不同 batch，但是其它维度位置相同的数据。例如 (n, c, h, w) 的四维 tensor，32bit 的低 16bit 放置 (0, ci, hi, wi) 的数据，高 16bit 放置 (1, ci, hi, wi) 的数据。

BM_STORE_4N 只用于 BM_INT8/BM_UINT8，表示一个 32bit 的数据空间将放置 4 个不同 batch，但是其它维度位置相同的数据，例如 (n, c, h, w) 的四维 tensor，32bit 的 0~7bit

放置 (0, ci, hi, wi) 的数据，8~15bit 放置 (1, ci, hi, wi) 的数据，16~23bit 放置 (2, ci, hi, wi) 的数据，24~31bit 放置 (3, ci, hi, wi) 的数据。

### 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 表示 tensor 的 shape，目前最大支持 8 维的 tensor。其中 num_dims 为 tensor 的实际维度数，dims 为各维度值，dims 的各维度值从 [0] 开始，比如 (n, c, h, w) 四维分别对应 (dims[0], dims[1], dims[2], dims[3])。

如果是常量 shape，初始化参考如下：

```
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]
};
```

bmrt_shape 接口可以设置 bm_shape_t，如下：

```
/*
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 可以得到 shape 的元素个数。接口声明如下：

```
/*
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);
```

比如 num_dims 为 4，则得到的个数为 dims[0]*dims[1]*dims[2]*dims[3]；如果 num_dims 为 0，则返回 1。

bmrt_shape_is_same 接口判断 2 个 shape 是否一样。接口声明如下：

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

一样返回 true；不一样返回 false。

只有 num_dims 以及对应的 dims[0]、dims[1]、…、dims[num_dims-1] 都一样，接口才认为是一样的 shape。

### Tensor

bm_tensor_t 结构体用来表示一个 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;
```

bmrt_tensor 接口可以配置一个 tensor。接口声明如下：

```
/*
This API is to initialize the tensor. It will alloc device mem to tensor->device_mem,
so user should 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);
```

bmrt_tensor_with_device 接口用已有的 device mem 配置一个 tensor。接口声明如下：

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

(续下页)

(接上页)

```
void bmrt_tensor_with_device(bm_tensor_t* tensor, bm_device_mem_t device_mem,
                             bm_data_type_t dtype, bm_shape_t shape);
```

这里 bmrt_tensor 和 bmrt_tensor_with_device 接口是为了方便用户初始化一个 tensor，用户也可以自己对 bm_tensor_t 每个成员进行初始化，不借助任何接口。

bmrt_tensor_bytesize 用于得到 tensor 的大小，单位是字节，它用通过 tensor 的元素个数乘以数据类型的字节数得到。接口声明如下：

```
/*
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 用于得到 device mem 的大小，单位是字节。接口声明如下：

```
/*
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);
```

创建 bmruntime，返回 runtime 指针。其他接口 (bmrt_xxxx 类接口)，需要的句柄都是该 runtime 指针。

### 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);
```

创建 bmruntime，支持传入多个 bm_handle，用于运行分布式的 bmodel。

### bmrt_destroy

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

销毁 bmruntime，释放资源。

用户通常开始创建 runtime，退出前销毁 runtime，举例如下：

```
// 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);
```

从 runtime 指针中得到设备句柄 bm_handle，在 bm_xxxx 一类接口中需要用到。

### 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);
```

加载 bmodel 文件，加载后 bmruntime 中就会存在若干网络的数据，后续可以对网络进行推理。

### 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);
```

加载 bmodel，不同于 bmrt_load_bmodel，它的 bmodel 数据存在内存中。

### bmrt_show_neuron_network

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

打印 bmruntime 中存在的网络的名称。

### 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);
```

获得 bmruntime 中存在的网络的数量。

### 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 declare 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);
```

得到 runtime 中存在的所有网络的名称。该接口会为 network_names 申请内存，所以不再使用的时候需要调用 free 释放它。

使用方法举例如下：

```
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

网络的信息表示如下：

```
/* 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] */
  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 表示一个网络的全部信息，bm_stage_info_t 表示该网络支持的不同的 shape 情况。

input_num 表示输入的数量，input_names/input_dtypes/input_scales 以及 bm_stage_info_t 中的 input_shapes 都是这个数量。

output_num 表示输出的数量，output_names/output_dtypes/output_scales 以及 bm_stage_info_t 中的 output_shapes 都是这个数量。

input_scales 和 output_scales 只有整型时有用；浮点型时为默认值 1.0。

max_input_bytes 表示每个 input 最大的字节数，max_output_bytes 表示每个 output 最大的字节数。每个网络可能有多个 stage，用户可能需要申请每个 input/output 的最大字节数，存放各种 stage 的数据。

input_zero_point 和 output_zero_point 记录在非对称量化 int8 网络的情况下输入和输出的 zero_point 值。

input_loc_devices 和 output_loc_devices 记录在分布式网络的情况下输入和输出设备号。

core_num 记录网络所需的 core 数量。

addr_mode 记录网络的地址分配模式，0 表示基础模式，1 表示 io_alone 模式，2 表示 io_tag 模式，3 表示 io_tag_fuse 模式。

bmrt_get_network_info 根据网络名，得到某个网络的信息，接口声明如下：

```
/*
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

打印网络的信息，主要在调试中需要用到，接口声明如下：

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

### bmrt_launch_tensor

对指定的网络，进行推理。接口声明如下：

```
/*
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 processor
↪program will not be blocked
if the neuron network is static-compiled and has no cpu layer. Otherwize, the host processor
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].
```

(续下页)

(接上页)

```
                        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);
```

用户在推理前需要初始化网络需要的 input_tensors，包括 input_tensors 中的数据。output_tensors 用于返回推理的结果。

**需要注意:**

- 该接口会为 output_tensors 申请 device mem，用于存储结果数据。当用户不再需要结果数据的时候，需要主动释放 device mem。
- 推理结束后，输出数据是以 BM_STORE_1N 存储；输出的 shape 存储在每个 output_tensor 的 shape 中。
- 该接口为异步接口，用户需要调用 bm_thread_sync 确保推理完成。

使用方法举例如下:

```
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

对指定的网络，进行推理。接口声明如下：

```
/*
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].
                                   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.
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);
```

与 bmrt_launch_tensor 不同的地方在于，用户可以在 output_tensors 中指定输出的 device mem，以及输出的 store mode。

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

具体说明如下：

- 当 user_mem 为 false 时，接口会为每个 output_tensor 申请 device mem，并保存输出数据。
- 当 user_mem 为 true 时，接口不会为 output_tensor 申请 device mem，用户需要在外部申请，申请的大小可以通过 bm_net_info_t 中的 max_output_bytes 指定。
- 当 user_stmode 为 false 时，输出数据以 BM_STORE_1N 排列。
- 当 user_stmode 为 true 时，输出数据根据各个 output_tensor 中的 st_mode 指定。

- 当深度学习处理器硬件架构支持多核时，该接口默认使用从 core0 开始的 N 个 core 来做推理，如果需要指定使用具体的深度学习处理器 core，需要使用 bmrt_launch_tensor_multi_cores 来完成。N 由当前 bmodel 决定。

**需要注意:** 该接口为异步接口，用户需要调用 bm_thread_sync 确保推理完成。

使用方法举例如下:

```
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,
                                 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

对指定的网络，进行 npu 推理。接口声明如下:

```
/*
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 support multiple inputs, and multi thread safety.
```

(续下页)

(接上页)

```
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 don'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] have been allocated memory.
                                false: output_datas[i] have 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[],
                bm_shape_t output_shapes[], int output_num, bool user_mem);
```

与 bmrt_launch_tensor 不同的地方在于:

- 输入和输出都存储在系统内存。
- 为同步接口。接口返回的时候推理已经完成。

### bmrt_launch_tensor_multi_cores

对指定的网络，选择指定的深度学习处理器 core 推理。接口声明如下:

```
/*
To launch the inference of the neuron network with setting input tensors, and support multi core inference.
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 not
be blocked. bm_thread_sync_from_core should be called to make sure inference is finished.
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.
```

(续下页)

(接上页)

```
                        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.
        [in] core_list        core id list those will be used to inference
        [in] core_num           number of the core list
Returns:    bool - true: Launch success. false: Launch failed.
*/
bool bmrt_launch_tensor_multi_cores(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,
                            const int *core_list, int core_num);
```

具体说明如下：

- 该函数可以选择推理时的深度学习处理器 core，仅对于支持多核深度学习处理器的硬件架构有效。其余参数使用同 bmrt_launch_tensor_ex 接口。

**需要注意:** 该接口为异步接口，用户需要调用 bm_thread_sync_from_core 确保推理完成。

#### bmrt_pre_alloc_neuron_multi_cores

对指定的网络，预先申请深度学习处理器推理计算所需要的设备内存。接口声明如下:

```
/*
To pre-allocate the neuron network compute memory during multi-cores arch inference.
This API only used for multi-cores arch runtime, need call before bmrt_launch_tensor_multi_
↪cores API.
After calling this API, the memory during neuron network inference is pre-allocated, can reduceF
↪first bmrt_launch_tensor_multi_cores API time cost.
If no use this API, is also OK, bmrt will auto alloc compute memory during first launch tensor.

Parameters: [in] p_bmrt - Bmruntime that had been created.
        [in] net_name - The name of the neuron network.
        [in] stage_idx - Witch network stage need to be pre-allocate.
        [in] core_list        core id list those will be used to inference
        [in] core_num           number of the core list
```

(续下页)

(接上页)

```
Returns:    bool - true: Pre-allocate success. false: Pre-allocate failed.
*/
bool bmrt_pre_alloc_neuron_multi_cores(void *p_bmrt, const char *net_name, int stage_idx,
                                       const int *core_list, int core_num);
```

具体说明如下：

- 该函数仅对于支持多核深度学习处理器的硬件架构有效，可以减少第一次调用 bmrt_launch_tensor_multi_cores 接口时的时间。
- 默认不使用该函数的情况下，在指定模型第一次调用 bmrt_launch_tensor_multi_cores 时会自动地花费时间申请深度学习处理器推理计算所需要的设备内存。

### 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);
```

该接口用于 DEBUG。它会校验 runtime 的数据，打印 runtime 的一些信息，方便调试。

### get_bmodel_api_info_c

```
/*
 * This API only supports the neuron nework that is static-compiled.
 * After calling this API, api info will be setted and return,
 * and then you can call `bm_send_api` to start deep-learning processor inference.
 * When you no longer need the memory, call bmrt_free_api_info to avoid memory leaks.
 *
 * @param [in]    p_bmrt           Bmruntime that had been created
 * @param [in]    net_name         The name of the neuron network
 * @param [in]    input_tensors    Array of input tensor, defined like bm_tensor_t input_
↪tensors[input_num],
 *                           User should initialize each input tensor.
 * @param [in]    input_num        Input number
 * @param [in]    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 F
↪true, this interface
 *                           will use device mem of output_tensors to store output data, and not F
↪alloc device mem;
 *                           Or it will alloc device mem to store output. If user_stmode is true, it F
↪will use stmode in
```

(续下页)

(接上页)

```
*                          each output tensor; Or stmode will be BM_STORE_1N as default.
* @param [in]    output_num       Output number
* @param [in]    user_mem         whether device_mem of output tensors are set
* @param [in]    user_stmode      whether stmode of output tensors are set
*/
api_info_c *get_bmodel_api_info_c(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);
```

- 该函数使用方法类似 bmrt_launch_tensor_ex，但是它只是返回 bmodel 推理前需要下发给深度学习处理器的推理信息，并不会启动推理。该函数返回的信息可以通过 bm_send_api 发送给深度学习处理器启动推理，因此 get_bmodel_api_info + bm_send_api 和 bmrt_launc_tensor_ex 作用是等价的。
- **在该 api_info 使用结束后需要调用 bmrt_free_api_info 来释放内存**。

#### bmrt_free_api_info

- 释放 api_info 所申请的内存空间。

### 3.1.3 C++ Interface

BMRuntime 的 C++ 语言接口，对应的头文件为 bmruntime_cpp.h，对应的 lib 库为 libbmrt.so。用户程序使用 C++ 接口时建议使用该接口，该接口支持多种 shape 的静态编译网络，支持动态编译网络。

C++ 接口命名空间为 bmruntime，由 3 个类和全局 API 组成：

- class Context：用于网络管理，包括加载网络模型，获取网络信息
- class Network：用于对 class Context 中某个具体网络进行推理
- class Tensor：由 class Network 自动生成，用于对 input tensors 和 output tensors 进行管理
- Global APIs：全局 API，用于获得 tensor 的字节大小、元素个数、比较 shape 是否一致等功能

声明如下：

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

### class Context

Context 用于网络管理，比如加载模型，可以加载 1 个到多个模型；获取网络信息，可以得到已经加载了的所有网络的名称，以及通过网络名获得某个具体网络的信息。

#### 构造函数与析构函数

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

Context 的构造函数和析构函数。

用户调用 c++ 接口时，首先需要创建一个 Context 实例，可以指定 devid 创建实例，默认使用设备号 0。

使用参考如下：

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

也可以传入 bm_handle 创建实例，其中 bm_handle 由 bm_dev_request 生成。这种方式下需要注意，在退出程序的时候先析构 Context，再调用 bm_dev_free 释放 bm_handle。

使用参考如下：

```
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);
```

加载 bmodel。

bmodel 可以用内存形式，也可以是文件形式。可以被多线程调用。加载成功，返回 BM_SUCCESS; 否则返回其他错误码。

可以连续加载多个模型，但是多个模型之间不能有重复的网络名，不然会加载失败。

使用参考如下：

```
bm_status_t status;
status = p_ctx->load_bmodel(p_net1, net1_size); // p_net1指向bmodel的内存buffer
assert(status == BM_SUCCESS);
status = p_ctx->load_bmodel("net2.bmodel"); // 指定加载bmodel的文件路径
assert(status == BM_SUCCESS);
```

### get_network_number

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

获得已加载的网络的数量。

每个 bmodel 都含有 1 到多个网络，每次加载 bmodel，就会增加网络的数量。

### get_network_names

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

获得已加载的网络的名称，保存到 names 中。注意 vector 会先被 clear，再依次 push_back 网络名。

使用参考如下：

```
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;
```

通过网络名，获得某个具体网络的信息。

如果 net_name 存在，则返回 bm_net_info_t 的网络信息结构指针，内容包括它的输入输出的数量、名称、类型等等，具体参见 bm_net_info_t 结构体；如果 net_name 不存在，则返回 NULL。

使用参考如下：

```
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;
```

得到 context 的设备句柄，与构造函数传入的 bm_handle 是同一个，在调用 bm_xxxx 类接口时需要用到。

### trace

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

该接口用于 DEBUG。它会校验 context 的数据，打印 context 的一些信息，方便调试。

### class Network

Network 类用于对某个具体网络进行推理，该网络是从 Context 类已加载的网络中选取。该类会自动为该网络申请输入输出的 device memory。如果用户需要自己的 device memory，也可以在输入输出的 tensor 中设置。

#### 构造函数与析构函数

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

Network 的构造函数与析构函数。

ctx 为前文所述的 Context 实例，net_name 是 ctx 中已经加装的网络的名称，通过 net_name 创建一个 Network 实例。

stage_id 是指使用该网络的 stage 的次序号，如果为-1，则表明用户打算自己 Reshape 各个输入 tensor 的 shape；如果是具体 stage 的次序号，则 Network 的 input tensors 固定为这个 stage 的 shape，后续不能被 Reshape。

使用参考如下：

```
//net1, input tensors的shape后续可以被Reshape
Network net1(*p_ctx, "net1");
//net2, 采用bm_net_info_t中stage[1]的shape，后续不会被Reshape
Network net2(*p_ctx, "net2", 1);
```

#### Inputs

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

得到所有 input tensors。

用户在对该网络进行推理前，需要先得到 input tensors，然后对所有的 input tensor 进行设定，比如设置它的 shape，以及它的 data，也可以指定它的 device mem。

使用参考如下：

```
// 对net1的inputs初始化，假定它有2个input
auto &inputs = net1.Inputs();
inputs[0]->Reshape(shape0);
inputs[1]->Reshape(shape1);
// device_mem0和device_mem1已经存有需要的输入数据
inputs[0]->set_device_mem(device_mem0);
inputs[1]->set_device_mem(device_mem1);

// 对net2的inputs初始化，假定它有1个input
auto &inputs = net2.Inputs();
// inputs[0]->Reshape(shape0); // error，不能修改
// 假定需要的输入数据在系统内存，data0为数据指针
inputs[0]->CopyFrom(data0);
```

### Input

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

通过 input name 得到 input tensor。

### Forward

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

网络推理。

当 inputs 的数据都准备好后，就可以调用 Forward 进行推理。

sync 为 true 时，该接口会等待推理结束；sync 为 false 时，该接口为异步接口，接口退出的时候，推理正在进行中，不一定结束，需要调用 bm_thread_sync 接口确保它推理结束。

**特别注意**: 整个推理过程都在 device memory 上进行的，所以推理前输入数据必须已经存储在 input tensors 的 device mem 里面，推理结束后的结果数据也是保存在 output tensors 的 device mem 里面。

使用参考如下：

```
// net1进行推理
net1.Forward();
// net2进行推理
net2.Forward(false);
bm_thread_sync(p_ctx->hand());
```

### Outputs

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

得到 output tensors。

在 Forward 推理前，用户可以改变 output tensors 的 device_mem，以使推理结果保存在用户指定的 device mem 中；也可以改变 output tensors 的 store mode，以使推理结果以指定的 store mode 存放。

在 Forward 推理结束后，output tensors 里面的 shape 和 device_mem 中的数据才是有效的。

#### Output

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

通过 output name 得到 output tensor。

#### info

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

得到该网络的信息。

#### class Tensor

用于对网络的 input tensors 和 output tensors 进行管理。用户不能自己创建 Tensor，Tensor 在 Network 类生成时自动创建，所以构造函数和析构函数均不是 public。

#### CopyTo

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

将 tensor 的 device mem 上的数据拷贝到系统内存。

data 是指向系统内存数据的指针，size 用于指定拷贝的大小，offset 用于指定偏移。当 size 与 offset 不指定的时候，拷贝整个 tensor 的数据，也就是 ByteSize() 大小。

如果用户需要将输出结果拷贝到系统内存，则在推理结束后需要调用 CopyTo，把数据拷贝到系统内存。

#### CopyFrom

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

将系统内存的数据拷贝到 tensor 中的 device mem。

data 是指向系统内存数据的指针，size 用于指定拷贝的大小，offset 用于指定偏移。当 size 与 offset 不指定的时候，拷贝整个 tensor 的数据，也就是 ByteSize() 大小。

如果用户的输入在系统内存，则进行推理前需要调用 CopyForm，把数据拷贝到对应的 input tensor。

### Reshape

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

设置 tensor 的 shape。

主要是用来改变 input tensor 的 shape；对于 output tensor，该接口没有意义，因为它的 shape 是推理后得到的。

### ByteSize

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

获得 tensor 的数据的大小，以字节为单位。通过 (元素数量)*(元素类型的字节数) 计算得到。

### num_elements

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

获得 tensor 的元素数量。通过 dims[0] * dims[1] * ···* dims[num_dims-1] 计算得到；如果 num_dims 为 0，则返回 1 (说明该 Tensor 是一个标量)。

### tensor

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

获得 tensor 的 bm_tensor_t 结构，该结构中包含了 tensor 的 shape、data type、device mem、store mode。

### set_store_mode

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

设置 tensor 的 store mode。

在进行推理前，用户可以配置 input 的 store mode，用来表明输入数据的存储模式；也可以配置 output 的 store mode，用于表示推理后的数据存储模式。不配置情况下，默认为 BM_STORE_1N。

#### set_device_mem

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

设置 tensor 的 device mem。

在进行推理前，用户可以设置 input 的 device mem，表明输入数据的存储位置；也可以设置 output 的 device mem，用于指定输出的存储位置。

用户不设置的情况下，输入输出都存储在 Network 自动申请的 device mem 中。

另外用户配置的 device mem 的大小，不能小于 ByteSize()，不然因无法存储下整个 tensor 的数据而返回错误。

### Global APIs

#### ByteSize

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

获取数据类型的字节大小，比如 BM_FLOAT32 字节大小为 4，BM_INT8 字节大小是 1。

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

获得 device mem 的字节大小，也就是它的存储空间大小。

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

获得 bm_tensor_t 的字节大小，它等于 (tensor 的元素个数)*(tensor 的数据类型字节大小)。

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

获得 Tensor 的字节大小，它等于 (tensor 的元素个数)*(tensor 的数据类型字节大小)；等同于 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);
```

获得元素个数，也就是各个维度数量的乘积；如果 num_dims 为 0，则返回 1。

#### IsSameShape

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

比较两个 shape 是否一样，一样返回 true；不一样返回 false。

只有 num_dims 以及对应的 dims[0]、dims[1]、…、dims[num_dims-1] 都一样，接口才认为是一样的 shape。

### 3.1.4 Multi-thread Program

上文描述的 runtime 的 C 接口和 C++ 接口，都是线程安全的。但是兼容老版本而保留的其他接口 (Other Interface) 不一定是线程安全的，不推荐使用。

常用的使用方法有两种：

- 创建一个 bmruntime，加载所有的模型后，对各种网络进行多线程推理
- 每个线程创建一个 bmruntime，加载该线程需要的模型，进行网络推理

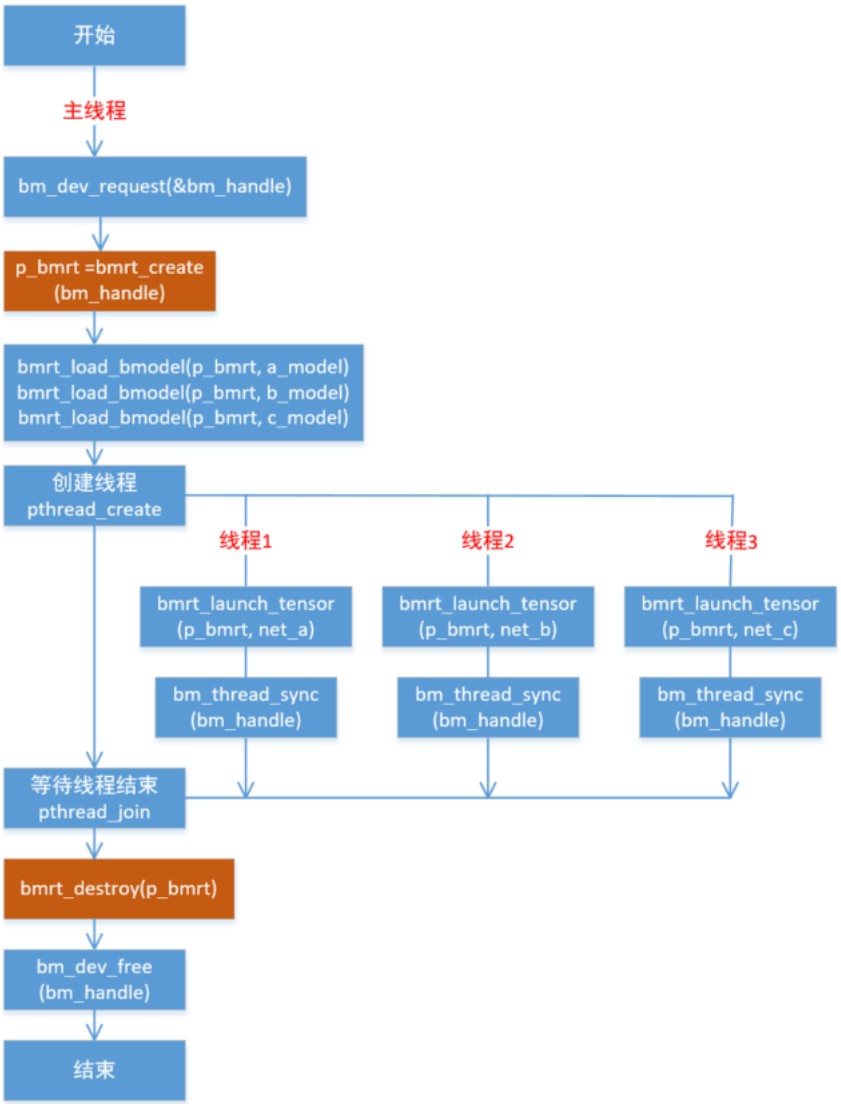
#### single runtime

单个 runtime，可以加载多个不同的模型，注意多个模型之间不能存在相同的网络，不然会认为冲突。同理，同样的模型只能加载一次。

可以通过这个 runtime，对已经加载的网络进行多线程推理，多个线程中的网络可以相同，也可以不同。

编程模型如下：




该图以 C 接口举例。

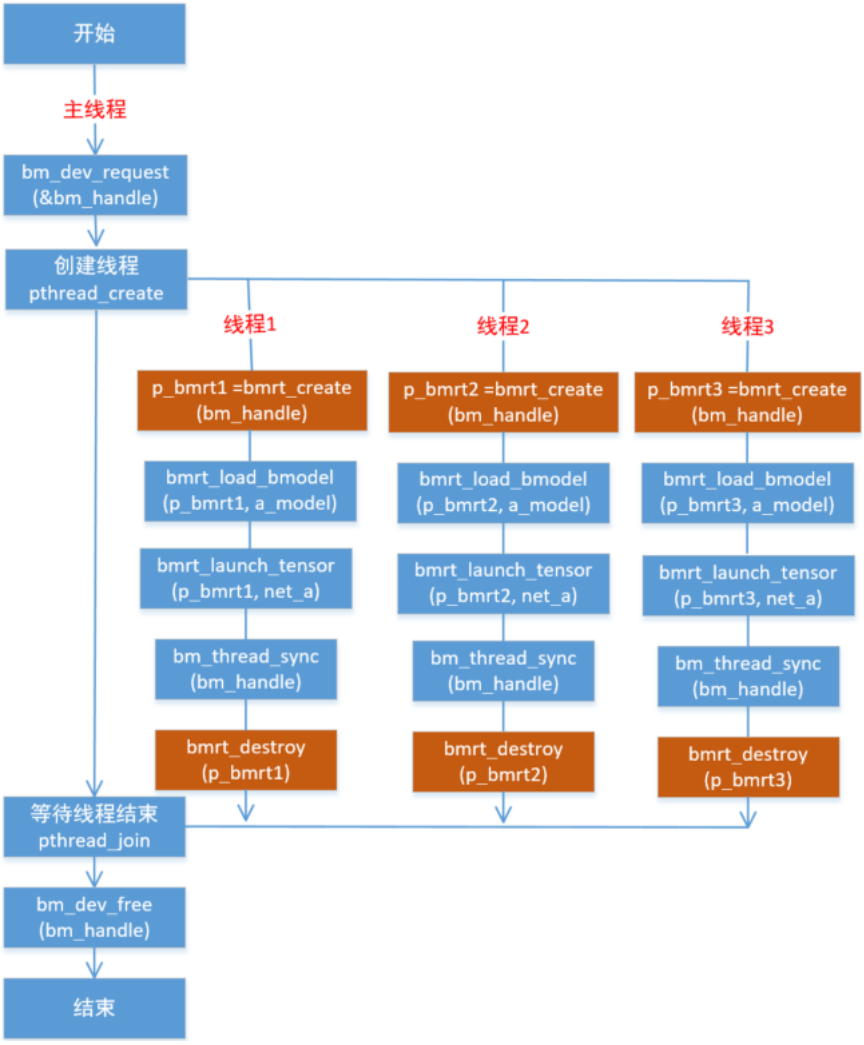
对于 C++ 接口，就是创建单个 Context 实例，然后通过 load_bmodel 加载网络模型。

然后在多个线程内分别创建 Network 实例进行推理，同样的 Network 实例的网络可以相同，也可以不同。

### multi runtime

用户可以创建多个线程，每个线程创建一个 bmruntime，各个 bmruntime 加载模型都是独立的，它们之间加载相同的模型。

编程模型如下：




该图以 C 接口举例。

对于 C++ 接口，就是各个线程分别创建一个 Context 实例，各自通过 load_bmodel 加载网络模型。

### how to choose

这两种多线程编程方式之间区别在于：

**单个 runtime，每个网络的 neuron 内存是只有一份**

所以使用单个 runtime 的时候，消耗的 neuron 内存少，但是如果对同一个网络进行多线程推理，就需要等待这个网络的 neuron 空间空闲；

而多个 runtime 的时候，每个 runtime 都加载相同的网络，跑同一个网络时不需要等待 neuron 空间空闲，但是消耗的 neuron 内存多。

根据用户的业务情况，可以用以下标准来选择使用哪一个方式：

**如果需要对同一个网络进行多线程推理，请使用多个 runtime；其他情况都使用单个 runtime**

CHAPTER 4

# BMRuntime 示例代码

## 4.1 BMRuntime 示例代码

### 4.1.1 Example with basic C interface

本节通过举例来介绍 runtime C 接口的使用，一个是通用的例子，可以在 PCIE 或者 SoC 上使用；另一个使用 mmap 方式，只能在 SoC 上使用

#### Common Example

范例说明：

- 创建 bm_handle 以及 runtime 实例
- 加载 bmodel，该模型中有一个 testnet 网络，有 2 个输入，2 个输出
- 准备 input tensors，包括每个 input 的 shape 和数据
- 启动推理运算
- 推理结束后将在 output_tensors 中结果数据拷贝到系统内存
- 退出程序前，释放 device mem、runtime 实例、bm_handle

需要注意，范例中将 output 的数据通过 bm_memcpy_s2d_partial 接口拷贝到系统内存，而不是用 bm_memcpy_s2d 接口，原因在于前者指定了 tensor 的大小，后者是按照 device mem 大小整个拷贝。output tensor 的实际大小是小于或等于 device mem 大小的，所以使用 bm_memcp_s2d 有可能内存溢出。所以推荐用 bm_memcpy_x2x_partial 接口，不要使用 bm_memcpy_x2x 接口。

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

  ret = bmrt_launch_tensor_ex(p_bmrt, "testnet", input_tensors, 2,
```

(续下页)

(接上页)

```
                        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 are 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

本例功能和上个例子相同，但没有进行 device mem 数据的拷入和拷出，而是采用 mmap 方式映射给应用程序直接访问。效率比上例高，但只能在 SoC 下使用。

```
#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);
```

(续下页)

(接上页)

```
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);
assert(BM_SUCCESS == status);
status = bm_mem_invalidate_device_mem(bm_handle, &output_tensors[1].device_mem);
assert(BM_SUCCESS == status);
// do other things
```

(续下页)

(接上页)

```
  // 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

本节通过举例来介绍 runtime C++ 接口的使用，一个是通用的例子，可以在 PCIE 或者 SoC 上使用；另一个使用 mmap 方式，只能在 SoC 上使用

#### Common Example

范例说明：

- 创建 bm_handle 以及 context 实例
- 加载 bmodel，该模型中有一个 testnet 网络，有 2 个输入，2 个输出
- 准备 input tensors，包括每个 input 的 shape 和数据
- 启动推理运算
- 推理结束后将在 output_tensors 中结果数据拷贝到系统内存
- 退出程序前，释放 bm_handle

对 Context 的 testnet 网络实例化了 2 个 Network，以此表明这几点：

- 当 Network 不指定 stage 的时候，每个 input 都需要 Reshape 来设置输入的 shape；当 Network 指定 stage 的时候，按照 stage 的 shape 来配置 input，不需要用户再 Reshape
- 同一个网络名，是可以被实例化成多个 Network，它们之间没有任何影响。同理每个 Network 可以多线程中推理

```cpp
#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 are system buffers to store result.
    auto &outputs = net1.Outputs();
    outputs[0]->CopyTo(output0);
    outputs[1]->CopyTo(output1);
    ..... // do other things
  }

  /*************************************************************/
  // 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();
```

(续下页)

(接上页)

```
    assert(BM_SUCCESS == status);
    // here all output info stored in output_tensors
    auto &outputs = net2.Outputs();
    ..... // do other things
  }
}
```

### MMAP Example

本例只实例化了一个 Network，主要是说明 mmap，如何使用。

```
#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);

  // 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);
```

(续下页)

(接上页)

```
  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 工具使用及 bmodel 验证

## 5.1 bmrt_test 工具使用及 bmodel 验证

### 5.1.1 bmrt_test 工具

bmrt_test 是基于 bmruntime 接口实现的对 bmodel 的正确性和实际运行性能的测试工具。包含以下功能:

1. 直接用随机数据 bmodel 进行推理，验证 bmodel 的完整性及可运行性
2. 通过固定输入数据直接用 bmodel 进行推理，会对输出与参考数据进行比对，验证证数据正确性
3. 测试 bmodel 的实际运行时间
4. 通过 bmprofile 机制进行 bmodel 的 profile

### 5.1.2 bmrt_test 参数说明

### 5.1.3 bmrt_test 输出

```
[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
```

**主要关注点：**

(1) 模型的纯推理时间，不包含加载输入和获取输出时间

(2) 推理数据展示，如开启比对，会显示比对成功等式信息

(3) 用 s2d 加载输入数据时间，通常前处理会将数据直接放到设备上，没有这个时间消耗

(4) 用 d2s 取出输出数据时间，通常表示在 pcie 上的数据传输时间，在 soc 上可以用 mmap，会更快

### 5.1.4 bmrt_test 常用方法

```
bmrt_test --context_dir bmodel_dir  # 运行bmodel，并比对数据，bmodel_
↪dir中要包含compilation.bmodel/input_ref_data.dat/output_ref_data.dat
bmrt_test --context_dir bmodel_dir  --compare=0 # 运行bmodel，bmodel_
↪dir中要包含compilation.bmodel
bmrt_test --bmodel xxx.bmodel # 直接运行bmodel，不比对数据
bmrt_test --bmodel xxx.bmodel --stage_idx 0  --shapes "[1,3,224,224]" #
↪运行多stage的bmodel模型，指定运行stage0的bmodel
bmrt_test --bmodel xxx.bmodel --core_list 0,1 #
↪在深度学习处理器core0和core1上同时运行该bmodel,注意该bmodel为多核编译且能够架构支持多核运行,core_
↪list中的值至少为0且不能大于深度学习处理器core数量-1
# 以下命令是通过环境变量使用bmruntime提供的功能，其他应用程序也可以使用
BMRUNTIME_ENABLE_PROFILE=1 bmrt_test --bmodel xxx.bmodel #
↪生成profile数据：bmprofile_data-x
BMRT_SAVE_IO_TENSORS=1 bmrt_test --bmodel xxx.bmodel  #
↪将模型推理的数据保存成input_ref_data.dat.bmrt和output_ref_data.dat.bmrt
```

### 5.1.5 比对数据生成与验证举例

1. 模型编译完成后，进行比对运行

   编译模型时在 deploy 阶段要添加--test_input 和 --test_reference，就会在编译输出文件夹中生成 input_ref_data.dat 和 output_ref_data.dat 文件

   接着执行‘bmrt_test --context_dir bmodel_dir’，便可验证模型推理数据正确性

2. pytorch 原始模型和编译出的 bmodel 数据比对

   将 pytorch 模型的输入 input_data 和输出 output_data 转为 numpy 的 array(torch 的 tensor 可以用 tensor.numpy())，然后存文件 (见以下代码)

```
# 单输入和单输出情况
input_data.astype(np.float32).tofile("input_ref_data.dat") #F
↪astype要根据bmodel的输入数据类型转换
output_data.astype(np.float32).tofile("output_ref_data.dat") #F
↪astype要根据bmodel的输出数据类型转换

# 多输入和多输出情况
with open("input_ref_data.dat", "wb") as f:
    for input_data in input_data_list:
        f.write(input_data.astype(np.float32).tobytes()) #F
↪astype要根据bmodel的输入数据类型转换
with open("output_ref_data.dat", "wb") as f:
    for output_data in output_data_list:
        f.write(output_data.astype(np.float32).tobytes()) #F
↪astype要根据bmodel的输出数据类型转换
```

   把生成的 input_ref_data.dat 和 output_ref_data.dat 放到 bmodel_dir 文件夹下然后‘bmrt_test --context_dir bmodel_dir’，看看结果是否出比对错误

### 5.1.6 常见问题

1. 编译模型时出现数据比对错误?

我们 bmcompiler 内部用的是 0.01 作为比对阈值，在少数情况下可能会超出范围而报错。

如果某一层实现的有问题，会出现成片比对错误，这时需要向我们开发人员反馈。

如果随机位置的零星错误，可能是个别值计算误差引起的。原因是编译时用的是随机数据，不排除会出现这种情况，所以建议编译时先加上 --cmp 0，在实际业务程序上验证结果是否正确

还有一种可能是网络中存在随机算子 (如 uniform_random) 或者排序算子 (如 topk、nms、argmin 等), 由于在前面计算过程会产生输入数据的浮点尾数误差，即使很小也会导致排序结果的 index 不同。这种情况下，可以看到比对出错的数据顺序上有差异，只能到实际业务上去测试