Classification Level: Top Secret( ) Secret( ) Internal( ) Public(√)

# RKNN API For RKNPU User Guide

(Technology Department, Graphic Compute Platform Center)

| | | |
|---|---|---|
| Mark:<br>[ ] Changing<br>[√] Released | Version: | 1.5.0 |
| | Author: | HPC/NPU Team |
| | Completed Date: | 22/May/2023 |
| | Reviewer: | Vincent |
| | Reviewed Date: | 22/May/2023 |

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

# Revision History

| Version | Modifier | Date | Modify Description | Reviewer |
|---|---|---|---|---|
| v0.6.0 | HPC Team | 2/Mar/2021 | Initial version | Vincent |
| v0.7.0 | HPC Team | 22/Apr/2021 | Remove description of swapping input data channels | Vincent |
| v1.0.0 | HPC Team | 30/Apr/2021 | Release version | Vincent |
| v1.1.0 | HPC Team | 13/Aug/2021 | 1. Add rknn_tensor_mem_flags<br>2. Add query commands for input/output Tensor native attributes<br>3. Add the memory layout of NC1HWC2 | Vincent |
| v1.2.0b1 | NPU Team | 07/Jan/2022 | 1. Add RK3588/RK3588s pla<br>2. Add rknn_set_core_mask interface<br>3. Add rknn_dump_context interface<br>4. Add details for input and output API | Vincent |
| v1.2.0 | HPC Team | 14/Jan/2022 | 1. Add the Glossary<br>2. Add the NPU SDK guide and building and compiling instruction<br>3. Add the section of how to do debugging<br>4. Add the description of C2 in the section NATIVE_LAYOUT | Vincent |
| v1.3.0 | NPU /HPC Team | 13/May/2022 | 1. Fix name from destory to destroy<br>2. Add RV1106/RV1103 user guide<br>3. Add details for NATIVE_LAYOUT<br>4. Add C API hardware platform support description<br>5. Add NPU version, utilization query and instruction of NPU power manual switch | Vincent |

| Version | Modifier | Date | Modify Description | Reviewer |
|---|---|---|---|---|
| v1.4.0 | NPU /HPC Team | 31/Aug/2022 | 1. For RV1106/RV1103 add rknn_create_mem_from_phys/ rknn_create_mem_from_fd/ rknn_set_weight_mem/ rknn_set_internal_mem support<br>2. Add function of weights shared<br>3.  Add support of utlization of SRAM for RK3588<br>4. Add the support of single batch multi-cores of NPU for RK3588<br>5. Add new version of driver supporting the query of frequency, voltage, setting of a delay time for shutdown , etc. | Vincent |
| v1.4.2 | HPC Team | 13/Feb/2023 | 1. Add RK3562 user guide<br>2. Add RKNN_FLAG_COLLECT_MODEL_INFO_ONLY flag description in rknn_init interface | Vincent |
| V1.5.0 | HPC Team | 22/May/2023 | 1. Add dynamic shape input API instructions and related data structure instructions<br>2. Add instructions for using Matmul API | Vincent |

# Table of Contents

# 1 Overview

RKNN SDK provides programming interfaces for all chips platforms with RKNPU, which can help users deploy RKNN models from RKNN-Toolkit2 and accelerate the implementation of AI applications.

# 2 Supported Hardware Platforms

This document applies to the following hardware platforms:

RK3566, RK3568, RK3588, RK3588S, RV1103, RV1106，RK3562

Note: RK3566_RK3568 is used to indicate RK3566/RK3568 in this document. RK3588 is used to indicate RK3588/RK3588S in this document.

# 3 Glossary

**RKNN Model**: It is the binary file running on the RKNPU, with the suffix .rknn.

**Inference with board**: It refer to use RKNN-Toolkit2 API interface to run the model. Actually, the model is running on the NPU on the development board.

**HIDL**: It denotes interface description language (IDL) to specify the interface between an android HAL and its users.

**CTS**: It is the Compatibility Test Suite from android automatic testing kit.

**VTS**: It is the Vendor Test Suite from android automatic testing kit.

**DRM**: It is the Direct Rendering Manager, a main stream of graph displaying framework on linux.

**tensor**: A multi-dimensional array of data.

**fd**: It is the file descriptor for representing the buffer.

**NATIVE_LAYOUT**: It refers to the native layout of NPU. In other words, it has the best performance when NPU is processing data with this memory layout.

**i8 model**: A quantized RKNN model which running by 8-bit signed integer data.

**fp16 model**: A non-quantized RKNN model which running by 16-bit half-float data.

# 4 Instructions

## 4.1 RKNN SDK Development Process

Before using the RKNN SDK, users first need to utilize the RKNN-Toolkit2 to convert the user's model to the RKNN model.

After getting the RKNN model file, users can choose using C interface to develop the application. The following chapters will explain how to develop applications via the RKNN SDK on RK3562/RK3566/RK3568/RK3588/RV1106/RV1103 platform.

## 4.2 RKNN Linux Platform Development Instructions

### 4.2.1 RKNN API Library For Linux

For the RK3562/RK3566/RK3568, the SDK library file is librknnrt.so under the directory of <sdk>/rknpu2/runtime. As for the RV1106/RV1103, the SDK library file is librknnmrt.so under the directory of <sdk>/rknpu2/runtime.

### 4.2.2 Example Usage

The SDK provides multiple reference examples such as MobileNet image classification, SSD object detection, YOLOv5 object detection, and dynamic shape input on the Linux platform.. These demos are the reference for developers to develop applications based on the RKNN SDK. The demo is located at the path of <sdk>/rknpu2/examples. Let's take RK3566/RK3568 rknn_mobilenet_demo as an example to explain how to get started quickly.

1) Compile Demo Source Code:

```
cd examples/rknn_mobilenet_demo
# set GCC_COMPILER in build-linux_RK3566_RK3568.sh to the correct compiler path
./build-linux_RK3566_RK3568.sh
```

2) Deploy to the RK3566/RK3568 device:

```
adb push install/rknn_mobilenet_demo_Linux /userdata/
```

3) Run Demo:

```
adb shell
cd /userdata/rknn_mobilenet_demo_Linux/
export LD_LIBRARY_PATH=./lib
./rknn_mobilenet_demo model/RK3566_RK3568/mobilenet_v1.rknn model/dog_224x224.jpg
```

## 4.3 RKNN Android Platform Development Instructions

### 4.3.1 RKNN API Library For Android

There are two ways to call the RKNN API on the Android platform:

1) The application can link librknnrt.so directly.

2) Application link to librknn_api_android.so implemented by HIDL on Android platform.

For Android devices that need to pass the CTS/VTS test, you can use the RKNN API based on the Android platform HIDL implementation. If the device does not need to pass the CTS/VTS test, it is recommended to directly link and use librknnrt.so as this way can shorten the callingand linking processing of each interface to offer better performance.

The code for the RKNN API implemented using Android HIDL is located in the vendor/rockchip/hardware/interfaces/neuralnetworks directory of the RK3566/RK3568 Android system SDK. When the Android system is compiled, some NPU related libraries will be generated (for applications only need to link librknn_api_android.so), as shown below:

```
/vendor/lib/librknn_api_android.so
/vendor/lib/librknnhal_bridge.rockchip.so
/vendor/lib64/librknn_api_android.so
/vendor/lib64/librknnhal_bridge.rockchip.so
/vendor/lib64/rockchip.hardware.neuralnetworks@1.0.so
/vendor/lib64/rockchip.hardware.neuralnetworks@1.0-adapter-helper.so
/vendor/lib64/hw/rockchip.hardware.neuralnetworks@1.0-impl.so
/vendor/bin/hw/rockchip.hardware.neuralnetworks@1.0-service
```

Alternatively, user can use the following command to recompile and generate the above library separately.

```
mmm vendor/rockchip/hardware/interfaces/neuralnetworks/ -j8
```

### 4.3.2 Example Usage

At present, the SDK provides multiple reference examples such as MobileNet image classification, SSD object detection, YOLOv5 object detection, Android APK with camera input, dynamic shape input, etc. The demo code is located in the <sdk>/rknpu2/examples directory. Users can use NDK to compile the demo executed in the Android command line. Let's take RK3566/RK3568 rknn_mobilenet_demo as an example to explain how to use this demo on the Android platform:

1) Compile Demo Source Code:

```
cd examples/rknn_mobilenet_demo
#set ANDROID_NDK_PATH under build-android.sh to the correct NDK path
./build-android_RK3566_RK3568.sh
```

2) Deploy to the RK3566 device:

```
adb push install/rknn_mobilenet_demo_Android /data/
```

3) Run Demo:

```
adb shell
cd /data/rknn_mobilenet_demo_Andorid/
export LD_LIBRARY_PATH=./lib
./rknn_mobilenet_demo model/RK3566_RK3568/mobilenet_v1.rknn model/dog_224x224.jpg
```

Note: The above demo uses the librknnrt.so by default. For developers who want to use librknn_api_android.so, please refer to the README.md under the path of examples/librknn_api_android_demo/README.md.

## 4.4 RKNN C API

### 4.4.1 C API hardware platform support description

（1）RKNN C API hardware platform support：

| | RKNN C API | RK3562/RK3566/<br>RK3568 | RK3588 | RV1106/RV1103 |
|---|---|---|---|---|
| 1 | rknn_init | √ | √ | √ |
| 2 | rknn_set_core_mask | × | √ | × |
| 3 | rknn_dup_context | √ | √ | × |
| 4 | rknn_destroy | √ | √ | √ |
| 5 | rknn_query | √ | √ | √ |
| 6 | rknn_inputs_set | √ | √ | × |
| 7 | rknn_run | √ | √ | √ |
| 8 | rknn_wait | × | × | × |
| 9 | rknn_outputs_get | √ | √ | × |
| 10 | rknn_outputs_release | √ | √ | √ |
| 11 | rknn_create_mem_from_mb_blk | × | × | × |
| 12 | rknn_create_mem_from_phys | √ | √ | √ |
| 13 | rknn_create_mem_from_fd | √ | √ | √ |
| 14 | rknn_create_mem | √ | √ | √ |
| 15 | rknn_destroy_mem | √ | √ | √ |
| 16 | rknn_set_weight_mem | √ | √ | √ |
| 17 | rknn_set_internal_mem | √ | √ | √ |
| 18 | rknn_set_io_mem | √ | √ | √ |
| 19 | rknn_set_input_shape | √ | √ | × |

For more detailed instructions on RKNN C API，please refer to section 4.4.3 API Reference.

（2）rknn_query params hardware platform support：

| | rknn_query params | RK3562/<br>RK3566/<br>RK3568 | RK3588 | RV1106/RV1103 |
|---|---|---|---|---|
| 1 | RKNN_QUERY_IN_OUT_NUM | √ | √ | √ |
| 2 | RKNN_QUERY_INPUT_ATTR | √ | √ | √ |
| 3 | RKNN_QUERY_OUTPUT_ATTR | √ | √ | √ |
| 4 | RKNN_QUERY_PERF_DETAIL | √ | √ | × |
| 5 | RKNN_QUERY_PERF_RUN | √ | √ | × |
| 6 | RKNN_QUERY_SDK_VERSION | √ | √ | √ |
| 7 | RKNN_QUERY_MEM_SIZE | √ | √ | √ |
| 8 | RKNN_QUERY_CUSTOM_STRING | √ | √ | √ |
| 9 | RKNN_QUERY_NATIVE_INPUT_ATTR | √ | √ | √ |
| 10 | RKNN_QUERY_NATIVE_OUTPUT_ATTR | √ | √ | √ |
| 11 | RKNN_QUERY_NATIVE_NC1HWC2_INPUT_ATTR | √ | √ | √ |
| 12 | RKNN_QUERY_NATIVE_NC1HWC2_OUTPUT_ATTR | √ | √ | √ |
| 13 | RKNN_QUERY_NATIVE_NHWC_INPUT_ATTR | √ | √ | √ |
| 14 | RKNN_QUERY_NATIVE_NHWC_OUTPUT_ATTR | √ | √ | √ |
| 15 | RKNN_QUERY_INPUT_DYNAMIC_RANGE | √ | √ | × |
| 16 | RKNN_QUERY_CURRENT_INPUT_ATTR | √ | √ | × |
| 17 | RKNN_QUERY_CURRENT_OUTPUT_ATTR | √ | √ | × |

### 4.4.2 API Process Description

At present, there are two groups of APIs that can be used on RK3562/RK3566/RK3568/RK3588, namely the general API interface and the API interface for the zero-copy process, but **RV1106/RV1103 only support the API interface for the zero-copy process**. The main difference between the two sets of APIs is that each time the general interface updates the frame data, the data allocated externally needs to be copied to the input memory of the NPU during runtime, while the interface of the zero-copy process uses pre-allocated memory directly (including memory created by NPU or other frame, such as DRM), reducing the cost of copying memory. When the user input data has only an virtual address, only the common API interface can be used; when the user input data has a physical address or fd, both sets of interfaces can be used. General API and zero-copy API interface cannot be called together.

For the general API interface, the *rknn_input* structure is initialized at first. The frame data is contained in the structure. The input of models is set by using *rknn_inputs_set* function. After the completion of inference, the inference output can be acquired by the *rknn_outputs_get* function and performed for post-processing. The frame data must be updated before inference. The process of general API call is shown in Figure 4-1, and user behavior is indicated by the process in yellow font.

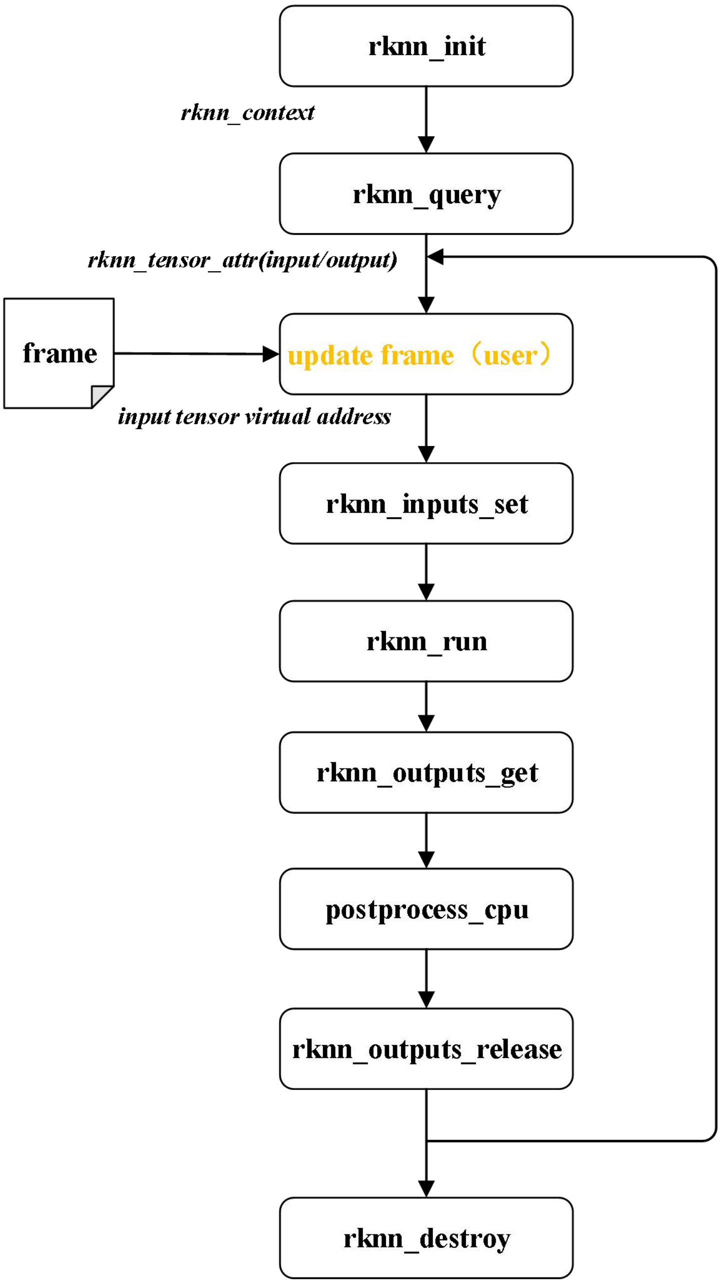

Figure 4-1: General process of calling general interface of API

For the zero-copy interface of API , after allocating memory, the *rknn_tensor_mem* structure is initialized by the memory information from allocated memory. Creating this structure before the inference is necessary. And the memory information can be retrieved by reading this structure after inference. According to whether the user needs to allocate the modular memory inside model (input/output/weights/intermediate result) and the differences of memory representation (file descriptor/physical address, etc.), there are the following three typical zero-copy calling processes, as shown in Figure 4-2, Figure 4-3 and Figure 4-4 respectively. The red font indicates the interface and data structure specially added for zero copy and the italic indicates the data structure passed between interface calls.

**1) Input/Output memory is allocated by API during runtime**

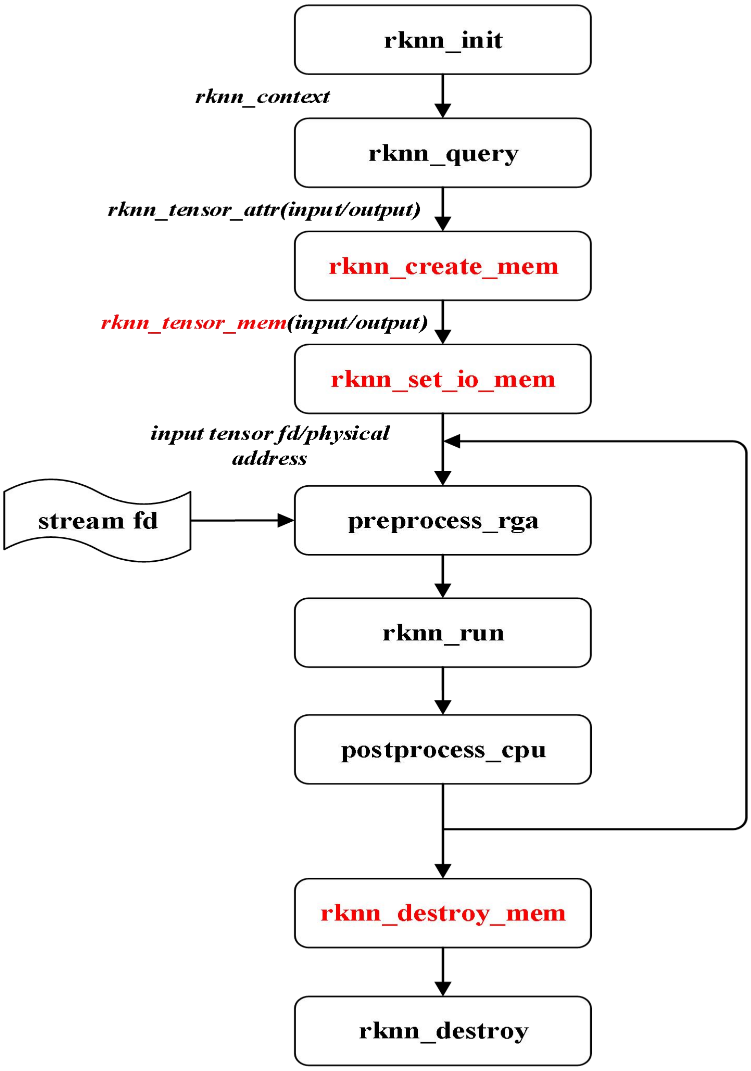

Figure 4-2: Process of zero-copy API interface call (input/output allocated internally)

As shown in Figure 4-2, the input/output memory information structure created by the rknn_create_mem interface includes the file descriptor and physical addresses. The RGA interface utilizes the memory information allocated by the NPU during runtime. In the above figure, preprocess_rga represents the RGA interface, and stream_fd represents the input source buffer represented by fd in RGA, postprocess_cpu represents the CPU implementation of post processing.

### 2) Input/output memory is allocated externally

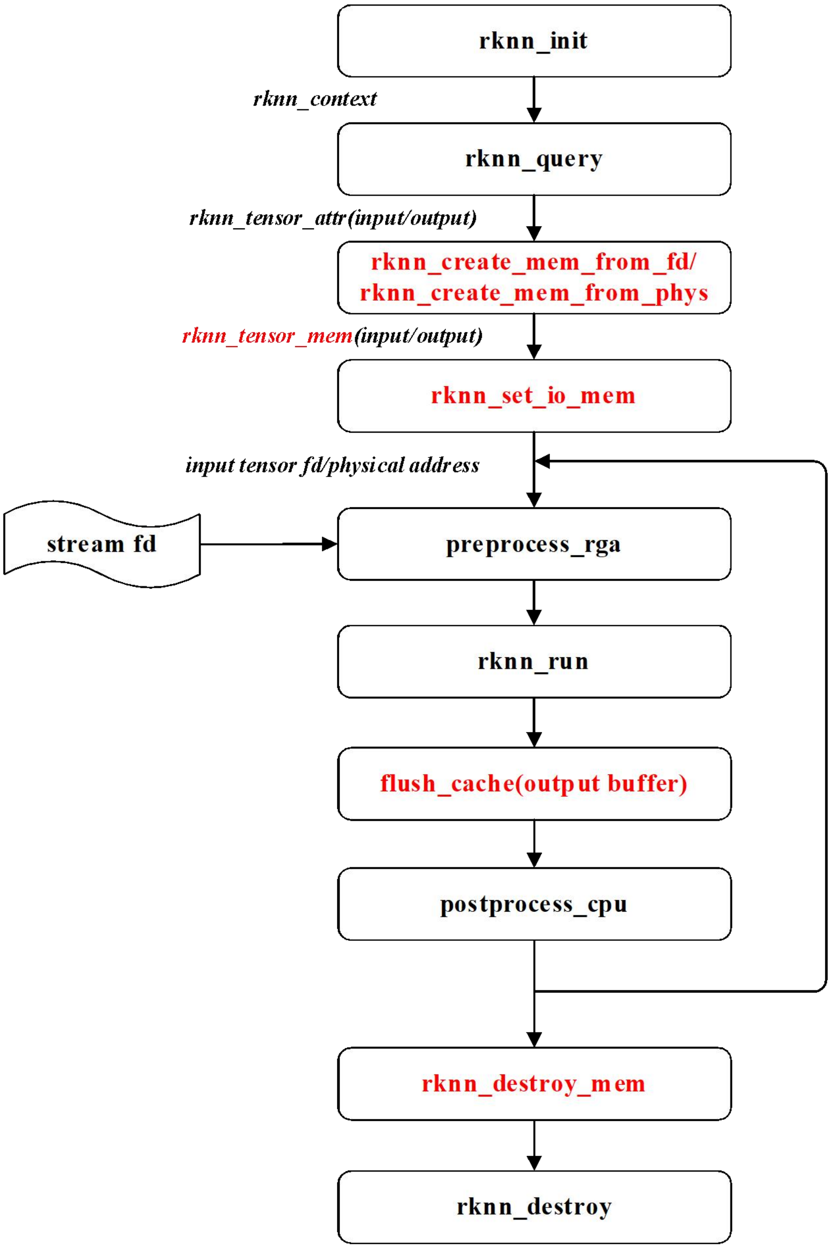

Figure 4-3: The process zero-copy API interface call (input/output allocated externally)

As shown Figure 4-3, flush_cache indicates that the user needs to call the interface associated with the memory type to flush the output buffer.

### 3) Input/output/weights/internal tensor memory is allocated externally

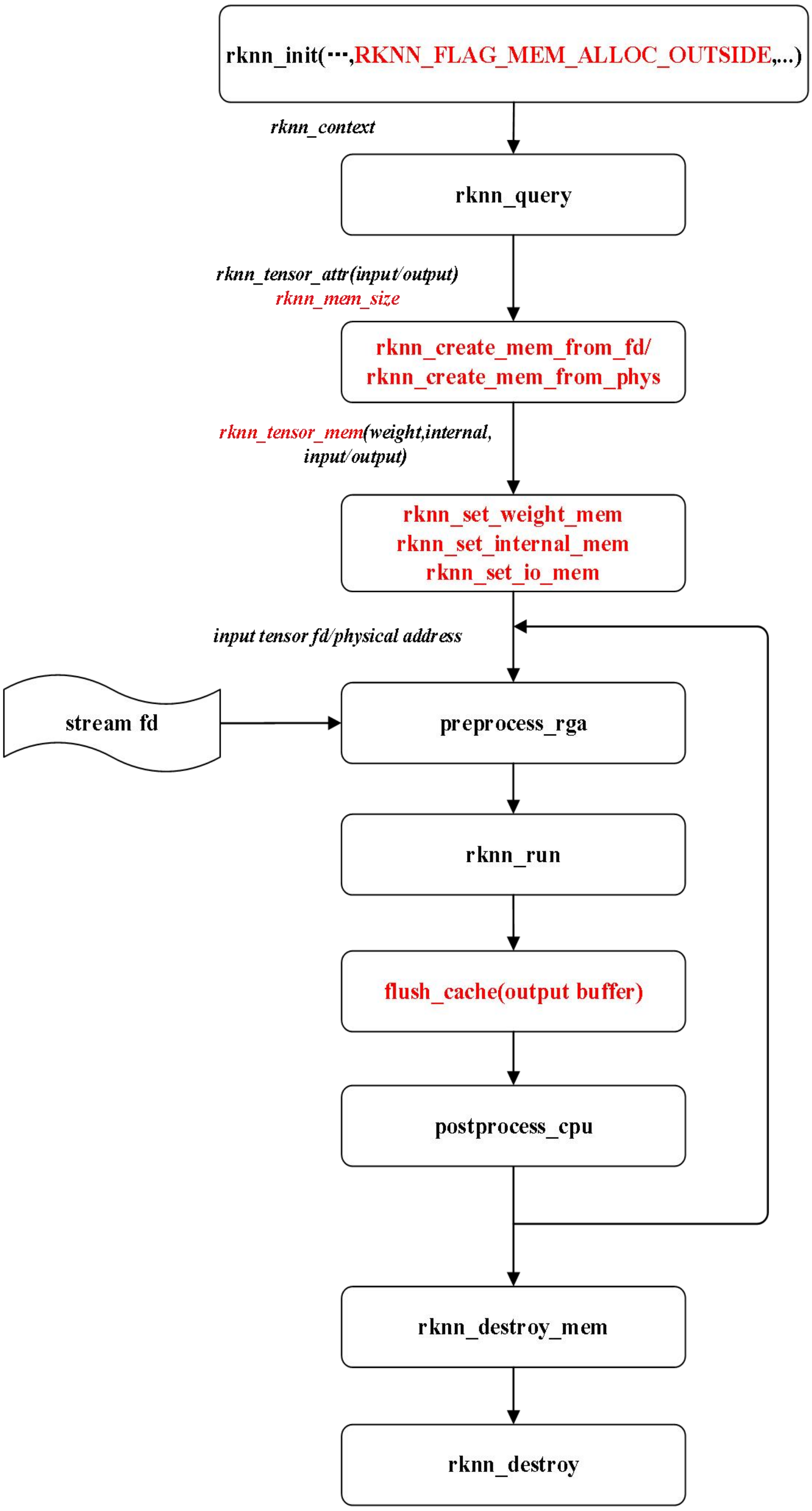

Figure 4-4: The process of zero-copy API interface call (input/output/weights/internal tensor allocated externally)

#### 4.4.2.1 API internal processing flow

When inferring the RKNN model, the original data need to go through three major processes: input processing, running model on NPU, and output processing. Currently, according to different model input formats and quantization methods, there are two different processes inside the general API interface (**Note: RV1106/RV1103 only supports int8 quantization model, and with uint8 data input with 1 or 3 or 4 channels**).

**1) int8 quantized model and the number of input channels is 1 or 3 or 4**

The processing flow of the original data is shown in Figure 4-5. Assuming that the input is a 4-channel model, the user must ensure that the color order of the R, G, and B channels is consistent with the training model. Normalization, quantization, and model inference are all running on the NPU. The output data layout format and dequantization process is running on the CPU.

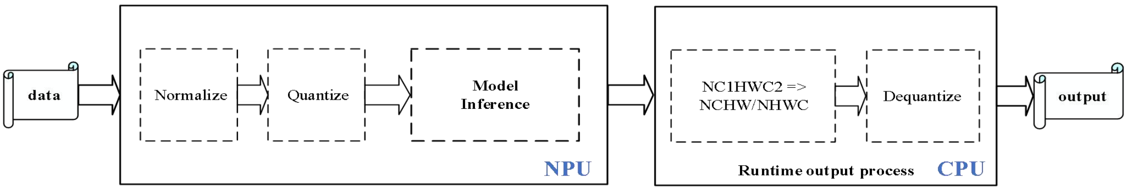

Figure 4-5: Optimized data processing flow

**2) int8 quantized model and non-quantized model when number of input channels is 2 or >= 4**

The flow of data processing is shown in Figure 4-6. The normalization, quantification, data layout format conversion, and dequantization of data are all running on the CPU except for the inference of the model which runs on the NPU. In this scenario, the processing efficiency of the input data flow would be lower than the optimized one in Figure 4-5.

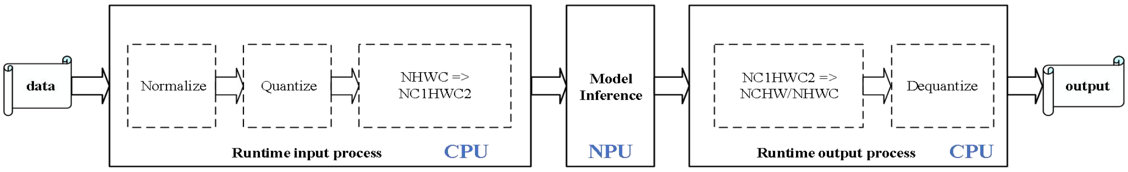

Figure 4-6: Common data processing flow

For the zero-copy API, there is only one processing flow for the internal process of the API, as shown in Figure 4-5. Conditions for using zero copy are listed below:

**1. The number of input channels is 1, 3 or 4.**

**2. The input width is 8 pixel aligned for RK3566/RK3568, The input width is 16 pixel aligned for RK3562, RK3588 and RV1106/RV1103 .**

**3. Int8 asymmetric quantized model.**

#### 4.4.2.2 Quantification and Dequantization

The quantization method, quantization data type and quantization parameters used in quantization and dequantization can be queried through the **rknn_query** interface.

At the moment, the NPU of RK3562/RK3566/RK3568/RV1106/RV1103 only supports asymmetric quantization, and does not support dynamic fixed-point quantization. The combination of data type and quantization method includes:

- int8 (asymmetric quantization)
- int16 (asymmetric quantization, **not yet implemented**)
- float16 (**unavailable RV1106/RV1103**)

Generally speaking, the normalized data is stored in 32-bit floating point data. For conversion of 32-bit floating point data to 16-bit floating point data, it is better for referring to the IEEE-754 standard. The following describes the quantization process, assuming that the normalized 32-bit floating point data is D.

**1) float32 to int8(asymmetric quantization)**

Assuming that the asymmetric quantization parameter of the input tensor is $S_q$, ZP, the data quantization process is expressed as the following formula:

$$D_q = \text{round}\left(\text{clamp}\left(D/S_q + \text{ZP},\text{-}128,127\right)\right)$$

In the above formula, clamp means to limit the value to a certain range. round means rounding processing.

**2) float32 to int16(asymmetric quantization)**

Assuming that the asymmetric quantization parameter of the input tensor is $S_q$, ZP, the data quantization process is expressed as the following formula:

$$D_q = \text{round}\left(\text{clamp}\left(D/S_q + \text{ZP},\text{-}32768,32767\right)\right)$$

The dequantization process is the inverse process of quantization, and the dequantization formula can be deduced according to the above quantization formula, which will not be repeated here.

#### 4.4.2.3 Dynamic shape input

Dynamic shape input means that the shape of the model input data can change at runtime (not supported by RV1106/RV1103). It can help to deal with the case where the input data size is not fixed and increase the flexibility of the model. Dynamic shape input plays an important role in image processing and sequence model inference. For details, please refer to "doc/RKNN_Dynamic_Shape_Usage.md". The requirements for the RKNN SDK version of this function are as follows:

**1. RKNN-Toolkit2 version >=1.5.0**

**2. RKNPU Runtime library version >=1.5.0**

The steps to use the dynamic shape input function are as follows:

**1. Confirm that the model supports dynamic shape input**

First, you need to confirm that your model supports dynamic shape input. Not all models support dynamic shape input. For example, the shape of a constant cannot be changed. RKNN-Toolkit2 returns an error to the model that cannot support dynamic shape input during the conversion process. The user can confirm whether the model supports dynamic shape input according to the error message. If your model does not support dynamic shape input, you need to retrain the model to support dynamic shape input.

**2. Create a dynamic shape input RKNN model**

Before using RKNN C API for inference, the model needs to be converted to RKNN format. The RKNN-Toolkit2 tool can be used to complete this process. If you wish to use dynamic shape inputs, set the list of shapes for each input during conversion. For a complete example of creating a dynamic shape input RKNN model, please refer to https://github.com/rockchip-linux/rknn-toolkit2/tree/master/examples/functions/dynamic_input.

**3. Call the C API to deploy the dynamic shape input RKNN model**

After gettting the dynamic shape input RKNN model, the next step is to use the RKNPU C API for deployment. Taking the common API as an example, the calling process is shown in Figure 4-7. After loading the dynamic shape input RKNN model, the input shape can be dynamically modified at runtime. First, the list of input shapes supported by the RKNN model can be queried through rknn_query. The shape list information supported by each input is returned in the form of the rknn_input_range structure, which includes the name of each input, data layout information, number of shapes, and specific shapes. Then, by calling the rknn_set_input_shape interface, passing in the rknn_tensor_attr pointer containing shape information can set the shape used by the current inference. After setting the input shape, you can call rknn_query again to query the input and output shapes after the current setting is successful. Finally, complete inference according to the normal API process. Every time you switch the input shape, you need to set a new shape again, prepare the data of the new shape size and call the rknn_inputs_set interface again..

**Note: For the dynamic shape input RKNN model, the rknn_query interface only supports non-NATIVE input/output attribute query commands, and does not support the zero-copy process in which users use external interfaces to allocate memory.**

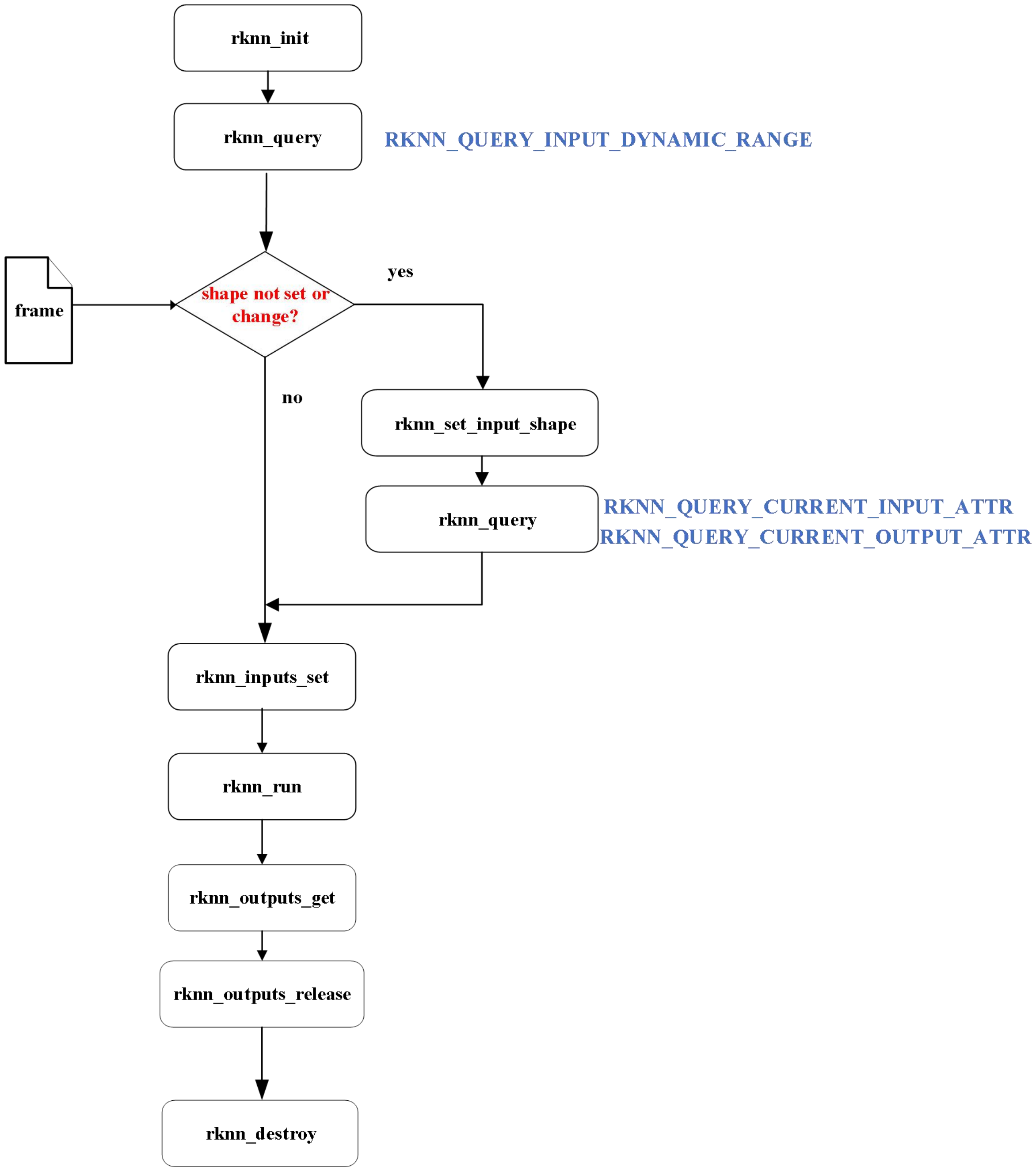

Figure 4-7 Dynamic shape input normal interface call process

For the zero-copy API, the normal usage call process is shown in Figure 4-8. First, the list of input shapes supported by the RKNN model can be queried through rknn_query, and the input and output memory of **the largest shape** can be allocated by calling the rknn_create_mem interface. Then, by calling the rknn_set_input_shape interface, passing in the rknn_tensor_attr pointer containing shape information can set the shape used by the current inference. After setting the input shape, you can call rknn_query again to query the successfully set input and output shapes. Finally, call the rknn_set_io_mem interface to set the required input and output memory. Every time you switch the input shape, you need to set a new shape again, prepare the data of the new shape size and call the rknn_set_io_mem interface again • .

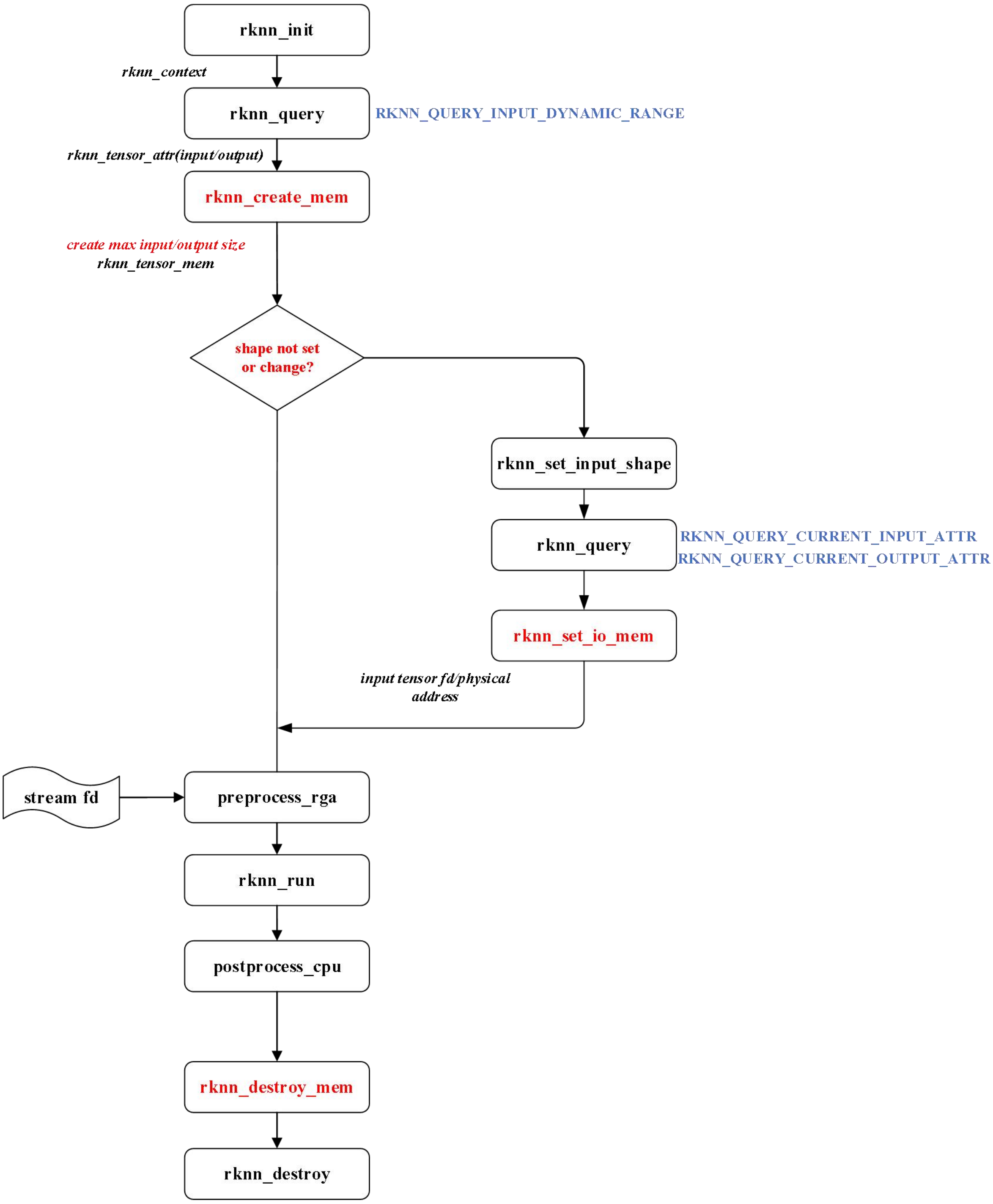

Figure 4-8 Zero-copy API call process of the dynamic shape input interface

The complete dynamic shape input C API Demo is located at https://github.com/rockchip-linux/rknpu2/tree/master/examples/rknn_dynamic_shape_input_demo.

### 4.4.3 API Reference

#### 4.4.3.1 rknn_init

The rknn_init will do the following things, such as, creating the *rknn_context* object, loading the RKNN model, and performing specific initialization behaviors according to the flag and *rknn_init_extend* structure.

| API | rknn_init |
|---|---|
| Description | Initialize rknn. |
| Parameters | *rknn_context *context*: The pointer of *rknn_context* object. After the function is called, the obejct of the context will be returned with the information. |
| | *void *model*: Binary data for the RKNN model or the path of RKNN model. |
| | *uint32_t size*: When model is stored in binary data, it indicates the size of the model. The size is 0 when model is given as the path. |
| | *uint32_t flag*: A specific initialization flag. Now, it only has the following flags.<br>RKNN_FLAG_COLLECT_PERF_MASK: For querying the time it took to run each particular layer during runtime;<br>RKNN_FLAG_MEM_ALLOC_OUTSIDE: Used to indicate that model inputs, outputs, weights, and intermediate tensor memory are all allocated by the user;<br>RKNN_FLAG_SHARE_WEIGHT_MEM：Used to share weight from another model;<br>RKNN_FLAG_COLLECT_MODEL_INFO_ONLY: Used to initialize an empty context, it can only call the rknn_query interface to query the total size of the model weight memory and the total size of the internal tensor, but can not perform inference. |
| | *rknn_init_extend*: The extended information during specific initialization. It is disabled at the moment, which indicates this must be passed by the NULL. If using share weight， it should pass the pointer of another rknn_context pointing to another model. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
rknn_context ctx;
int ret = rknn_init(&ctx, model_data, model_data_size, 0, NULL);
```

Explanation:

RKNN_FLAG_MEM_ALLOC_OUTSIDE:  Two main functions

**1)** All memory is allocated by users, which is handy for management of overall memory useage of the whole system.

**2)** For the case of reusing memory, in particular for the RV1103/1106, in which the memory usage is quite limited, the internal tensor memory from two different models can be resued only when these two models are running in series order (parallel running is not allowed in this cased) . This is same for more than two or three models for sharing internal tensor memory

The example code is shown below.

```
rknn_context ctx_a, ctx_b;

rknn_init(&ctx_a, model_path_a, 0, RKNN_FLAG_MEM_ALLOC_OUTSIDE, NULL);
rknn_query(ctx_a, RKNN_QUERY_MEM_SIZE, &mem_size_a, sizeof(mem_size_a));

rknn_init(&ctx_b, model_path_b, 0, RKNN_FLAG_MEM_ALLOC_OUTSIDE, NULL);
rknn_query(ctx_b, RKNN_QUERY_MEM_SIZE, &mem_size_b, sizeof(mem_size_b));

max_internal_size = MAX(mem_size_a.total_internal_size, mem_size_b.total_internal_size);
internal_mem_max = rknn_create_mem(ctx_a, max_internal_size);

internal_mem_a = rknn_create_mem_from_fd(ctx_a, internal_mem_max->fd,
        internal_mem_max->virt_addr, mem_size_a.total_internal_size, 0);
rknn_set_internal_mem(ctx_a, internal_mem_a);

internal_mem_b = rknn_create_mem_from_fd(ctx_b, internal_mem_max->fd,
        internal_mem_max->virt_addr, mem_size_b.total_internal_size, 0);
rknn_set_internal_mem(ctx_b, internal_mem_b);
```

**RKNN_FLAG_SHARE_WEIGHT_MEM**: It is used for models with various input length, especially for some natural language models. Due to the fact that NPU cannot support flexible input length, it is required to have multiple models difference only in input resolution. Thus, the weight can be shared between models since they all have the same weights.

The following are steps for showing how to use this flag:

Assuming A and B model,

1) Using RKNN-Toolkit2 for generating model A with the input resolution A.

2) Using RKNN-Toolkit2 for generating model B without weight with input resolution B, which can be done via remove_weight in rknn.config() in RKNN-Toolkit2. This can reduce

the size of model B.

3) Initialize the model A .

4) Initialize the model B with this flag.

5) Then following the instruction as described:

```
rknn_context ctx_a, ctx_b;
rknn_init(&ctx_a, model_path_a, 0, 0, NULL);

rknn_init_extend extend;
extend.ctx = ctx_a;
rknn_init(&ctx_b, model_path_b, 0, RKNN_FLAG_SHARE_WEIGHT_MEM, &extend);
```

#### 4.4.3.2 rknn_set_core_mask

This function sets the specific cores running inside the NPU. For now, it only works at the RK3588 (including 3 NPU cores). It will return the error code if it is set on RK3562/RK3566/RK3568/RV1106/RV1103.

| | |
|---|---|
| API | rknn_set_core_mask |
| Description | Set the cores for the NPU. |
| Parameters | *rknn_context context*: The object of rknn context. |
| | rknn_core_mask core_mask: The specification of NPU core setting. It has the following choices:<br>**RKNN_NPU_CORE_AUTO** : Referring to automatic mode, meaning that it will select the idle core inside the NPU;<br>**RKNN_NPU_CORE_0** : Running on the NPU0 core;<br>**RKNN_NPU_CORE_1**: Runing on the NPU1 core;<br>**RKNN_NPU_CORE_2**: Runing on the NPU2 core;<br>**RKNN_NPU_CORE_0_1**: Running on both NPU0  and NPU1 core simultaneously;<br>**RKNN_NPU_CORE_0_1_2**: Running on both NPU0, NPU1 and NPU2 simultaneously. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
rknn_context ctx;
rknn_core_mask core_mask = RKNN_NPU_CORE_0;
int ret = rknn_set_core_mask(ctx, core_mask);
```

For multi-cores mode (when enabling RKNN_NPU_CORE_0_1 and RKNN_NPU_CORE_0_1_2), the following ops have better acceleration :

Conv, DepthwiseConvolution, Add, Concat, Relu, Clip, Relu6, ThresholdedRelu. Prelu, LeakyRelu.

Other type of op will fallback to Core0 to continue running. In the future update, some ops like Pool or ConvTranpose will be supported.

#### 4.4.3.3 rknn_dup_context

The rknn_dup_context creates a new context object referring to the same model. The new context is useful in the condition where the weight of the model need to be reused during executing the same model on the multi-thread (**Unavailable for RV1106/RV1103**) .

| API | rknn_dup_context |
|---|---|
| Description | Creates a new context for the same model, to reuse the weight of the model. |
| Parameters | *rknn_context *context_in*: The pointer of *rknn_context* object. After the function is called, the obejct of the input context will be initialized. |
| | *rknn_context *context_out*: The pointer of a *rknn_context* output object where information about this new created object is returned. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
rknn_context ctx_in;
rknn_context ctx_out;
int ret = rknn_dup_context(&ctx_in, &ctx_out);
```

#### 4.4.3.4 rknn_destroy

This function is used to release the rknn_context and its related resources.

| API | rknn_destroy |
|---|---|
| Description | Destroy the *rknn_context* object and its related resources. |
| Parameters | *rknn_context context*: The *rknn_context* object that is going to be destroyed. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
rknn_context ctx;
int ret = rknn_destroy(ctx);
```

#### 4.4.3.5 rknn_query

The rknn_query function can query the information of models and SDK, including model input and output information, layer-by-layer running time, total model inference time, SDK version, memory usage information, user-defined strings and other information.

| API | rknn_query |
|---|---|
| Description | Query the information about the model and the SDK. |
| Parameters | *rknn_context context*: The object of rknn_contex. |
| | *rknn_query_cmd* : The query command. |
| | *void* info*: The structure object that stores the result of the query. |
| | *uint32_t size*: The size of the info structure object. |
| Return | int: Error code (See RKNN Error Code). |

Currently, the SDK supports following query commands:

| Query command | Return result structure | Function |
|---|---|---|
| RKNN_QUERY_IN_OUT_NUM | rknn_input_output_num | Query the number of input and output Tensors. |
| RKNN_QUERY_INPUT_ATTR | rknn_tensor_attr | Query the input Tensor attribute. |
| RKNN_QUERY_OUTPUT_ATTR | rknn_tensor_attr | Query the output Tensor attribute. |
| RKNN_QUERY_PERF_DETAIL | rknn_perf_detail | Query the running time of each layer of the network. It only works when the flag of **RKNN_FLAG_COLLECT_PERF_MASK** is set via using the rknn_init. |
| RKNN_QUERY_PERF_RUN | rknn_perf_run | Query the total time of inference (excluding time in setting input/output) in microseconds . |
| RKNN_QUERY_SDK_VERSION | rknn_sdk_version | Query the SDK version. |
| RKNN_QUERY_MEM_SIZE | rknn_mem_size | Query the memory size allocated for the weights and internal tensors in the network. |
| RKNN_QUERY_CUSTOM_STRING | rknn_custom_string | Query the user-defined strings in the RKNN model. |
| RKNN_QUERY_NATIVE_INPUT_ATTR | rknn_tensor_attr | When using the zero-copy API interface, it queries the native input Tensor attribute, which is the model input attribute directly read by the NPU. |
| RKNN_QUERY_NATIVE_OUTPUT_ATTR | rknn_tensor_attr | When using the zero-copy API interface, query the native output Tensor attribute, which is the model output attribute directly from the NPU. |
| RKNN_QUERY_NATIVE_NC1HWC2_INPUT_ATTR | rknn_tensor_attr | When using the zero-copy API interface, it queries the native input Tensor attribute, which is the model input attribute directly read by the NPU.it is same with RKNN_QUERY_NATIVE_INP |

| | | |
|---|---|---|
| | | UT_ATTR. |
| RKNN_QUERY_NATIVE_NC1HWC2_OUTPUT_ATTR | rknn_tensor_attr | When using the zero-copy API interface, it queries the native output Tensor attribute, which is the model input attribute directly read by the NPU.it is same with RKNN_QUERY_NATIVE_OUTPUT_ATTR. |
| RKNN_QUERY_NATIVE_NHWC_INPUT_ATTR | rknn_tensor_attr | When using the zero-copy API interface, it queries the native input Tensor attribute, which is the model input attribute directly read by the NPU.it is same with RKNN_QUERY_NATIVE_INPUT_ATTR. |
| RKNN_QUERY_NATIVE_NHWC_OUTPUT_ATTR | rknn_tensor_attr | When using the zero-copy API interface, it queries the native output NHWC Tensor attribute, which is the model input attribute directly read by the NPU. |
| RKNN_QUERY_INPUT_DYNAMIC_RANGE | rknn_input_range | When using the RKNN model that supports dynamic shapes, query the model to support input information such as the number of shapes, list, data layout and name corresponding to the shape. |
| RKNN_QUERY_CURRENT_INPUT_ATTR | rknn_tensor_attr | When using the RKNN model that supports dynamic shapes, query the input attributes used by the model for current inference. |
| RKNN_QUERY_CURRENT_OUTPUT_ATTR | rknn_tensor_attr | When using the RKNN model that supports dynamic shapes, query the output attributes used by the model for current inference. |

Next we will explain each query command in detail.

### 1） Query the SDK version

The RKNN_*QUERY_SDK_VERSION* command can be used to query the version information of the RKNN SDK. You need to create the *rknn_sdk_version* structure object first.

Sample Code:

```
rknn_sdk_version version;
ret = rknn_query(ctx, RKNN_QUERY_SDK_VERSION, &version,
                 sizeof(rknn_sdk_version));
printf("sdk api version: %s\n", version.api_version);
printf("driver version: %s\n", version.drv_version);
```

### 2） Query the number of input and output Tensor

The *RKNN_QUERY_IN_OUT_NUM* command can be used to query the number of model input and output Tensor. You need to create the *rknn_input_output_num* structure object first.

Sample Code:

```
rknn_input_output_num  io_num;
ret = rknn_query(ctx, RKNN_QUERY_IN_OUT_NUM, &io_num,
                          sizeof(io_num));
printf("model input num: %d, output num: %d\n", io_num.n_input,
                          io_num.n_output);
```

### 3） Query input Tensor attribute (for general API interface)

The *RKNN_QUERY_INPUT_ATTR* command can be used to query the attribute of the model input Tensor. You need to create the *rknn_tensor_attr* structure object first (**Note: the tensor queried byRV1106/RV1103 is the tensor originally entered as native**) .

Sample Code:

```
rknn_tensor_attr  input_attrs[io_num.n_input];
memset(input_attrs, 0, sizeof(input_attrs));
for (int i = 0; i < io_num.n_input; i++) {
    input_attrs[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_INPUT_ATTR, &(input_attrs[i]),
                              sizeof(rknn_tensor_attr));
}
```

### 4） Query output Tensor attribute (for general API interface)

The *RKNN_QUERY_OUTPUT_ATTR* command can be used to query the attribute of the model output Tensor. You need to create the *rknn_tensor_attr* structure object first.

Sample Code:

```
rknn_tensor_attr output_attrs[io_num.n_output];
memset(output_attrs, 0, sizeof(output_attrs));
for (int i = 0; i < io_num.n_output; i++) {
    output_attrs[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_OUTPUT_ATTR, &(output_attrs[i]),
                                            sizeof(rknn_tensor_attr));
}
```

**5） Query layer-by-layer inference time of model**

After the rknn_run interface is called, the *RKNN_QUERY_PERF_DETAIL* command can be used to query the layer-by-layer inference time in microseconds. The premise of using this command is that the flag parameter of the rknn_init interface needs to include the RKNN_FLAG_COLLECT_PERF_MASK flag.

Sample Code:

```
rknn_context ctx;
int ret = rknn_init(&ctx, model_data, model_data_size,
                                RKNN_FLAG_COLLECT_PERF_MASK, NULL);
...
ret = rknn_run(ctx,NULL);
...
rknn_perf_detail perf_detail;
ret = rknn_query(ctx, RKNN_QUERY_PERF_DETAIL, &perf_detail,
                                sizeof(perf_detail));
```

**6） Query total inference time of model**

After the rknn_run interface is called, the *RKNN_QUERY_PERF_RUN* command can be used to query the inference time of the model (not including setting input/output) in microseconds.

Sample Code:

```
rknn_context ctx;
int ret = rknn_init(&ctx, model_data, model_data_size, 0, NULL);
...
ret = rknn_run(ctx,NULL);
...
rknn_perf_run perf_run;
ret = rknn_query(ctx, RKNN_QUERY_PERF_RUN, &perf_run,
                    sizeof(perf_run));
```

7） **Query the memory allocation of the model**

After the rknn_init interface is called, when the user needs to allocate memory for network by themself, the *RKNN_QUERY_MEM_SIZE* command can be used to query the weights of the model and the internal memory (excluding input and output) in the network. The requirement of using this command is that the flag parameter of the rknn_init interface needs to enable the RKNN_FLAG_MEM_ALLOC_OUTSIDE flag.

Sample Code:

```
rknn_context ctx;
int ret = rknn_init(&ctx, model_data, model_data_size,
                    RKNN_FLAG_MEM_ALLOC_OUTSIDE , NULL);
rknn_mem_size mem_size;
ret = rknn_query(ctx, RKNN_QUERY_MEM_SIZE, &mem_size,
                    sizeof(mem_size));
```

**8） Query User-defined string in the model**

After the rknn_init interface is called, if the user has added custom strings when generating the RKNN model, the *RKNN_QUERY_CUSTOM_STRING* command can be used to retrieve user-defined strings.

Sample Code:

```
rknn_context ctx;
int ret = rknn_init(&ctx, model_data, model_data_size, 0, NULL);
rknn_custom_string custom_string;
ret = rknn_query(ctx, RKNN_QUERY_CUSTOM_STRING, &custom_string,
                    sizeof(custom_string));
```

9） **Query native input tensor attribute (for zero-copy API interface)**

The *RKNN_QUERY_NATIVE_INPUT_ATTR* command can be used to query the native attribute of the model input Tensor. User need to create the *rknn_tensor_attr* structure object first.

Sample Code:

```
rknn_tensor_attr input_attrs[io_num.n_input];
memset(input_attrs, 0, sizeof(input_attrs));
for (int i = 0; i < io_num.n_input; i++) {
    input_attrs[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_NATIVE_INPUT_ATTR,
                                            &(input_attrs[i]), sizeof(rknn_tensor_attr));
}
```

### 10） Query native output tensor attribute (for zero-copy API interface)

The *RKNN_QUERY_NATIVE_OUTPUT_ATTR* command can be used to query the native attribute of the model output Tensor. User need to create the *rknn_tensor_attr* structure object first.

Sample Code:

```
rknn_tensor_attr output_attrs[io_num.n_output];
memset(output_attrs, 0, sizeof(output_attrs));
for (int i = 0; i < io_num.n_output; i++) {
    output_attrs[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_NATIVE_OUTPUT_ATTR,
                                            &(output_attrs[i]), sizeof(rknn_tensor_attr));
}
```

### 11） Query native output tensor attribute (for zero-copy API interface)

The *RKNN_QUERY_NATIVE_NHWC_INPUT_ATTR* command can be used to query the NHWC attribute of the model input Tensor. User need to create the *rknn_tensor_attr* structure object first.

```
rknn_tensor_attr input_attrs[io_num.n_input];
memset(input_attrs, 0, sizeof(input_attrs));
for (int i = 0; i <sdk>/rknpu2/runtime io_num.n_input; i++) {
input_attrs[i].index = i;
ret = rknn_query(ctx, RKNN_QUERY_NATIVE_NHWC_INPUT_ATTR,
                                    &(input_attrs[i]), sizeof(rknn_tensor_attr));
}
```

### 12） Query native output tensor attribute (for zero-copy API interface)

The *RKNN_QUERY_NATIVE_NHWC_OUTPUT_ATTR* command can be used to query the NHWC attribute of the model output Tensor. User need to create the *rknn_tensor_attr* structure object first.

```
rknn_tensor_attr output_attrs[io_num.n_output];
memset(output_attrs, 0, sizeof(output_attrs));
for (int i = 0; i <sdk>/rknpu2/runtime io_num.n_output; i++) {
output_attrs[i].index = i;
ret = rknn_query(ctx, RKNN_QUERY_NATIVE_NHWC_OUTPUT_ATTR,
                                    &(output_attrs[i]), sizeof(rknn_tensor_attr));
}
```

**13) Query the dynamic input shape information supported by the RKNN model (Note: RV1106/RV1103 does not support this interface)**

After the rknn_init interface is called, pass in the *RKNN_QUERY_INPUT_DYNAMIC_RANGE* command to query the input shape information supported by the model, including the number of input shapes, the list of input shapes, the layout and name of the input shapes, and other information. The rknn_input_range structure object needs to be created first.

The sample code is as follows:

```
rknn_input_range dyn_range[io_num.n_input];
memset(dyn_range, 0, io_num.n_input * sizeof(rknn_input_range));
for (uint32_t i = 0; i < io_num.n_input; i++) {
  dyn_range[i].index = i;
  ret = rknn_query(ctx, RKNN_QUERY_INPUT_DYNAMIC_RANGE,
          &dyn_range[i], sizeof(rknn_input_range));
}
```

**14) Query the input dynamic shape currently used by the RKNN model**

After the rknn_set_input_shape interface is called, pass in the *RKNN_QUERY_CURRENT_INPUT_ATTR* command to query the input attribute information currently used by the model. The rknn_tensor_attr structure needs to be created first (Note: RV1106/RV1103 does not support this command).

The sample code is as follows:

```
rknn_tensor_attr cur_input_attrs[io_num.n_input];
memset(cur_input_attrs, 0, io_num.n_input * sizeof(rknn_tensor_attr));
for (uint32_t i = 0; i < io_num.n_input; i++) {
  cur_input_attrs[i].index = i;
  ret = rknn_query(ctx, RKNN_QUERY_CURRENT_INPUT_ATTR,
          &(cur_input_attrs[i]), sizeof(rknn_tensor_attr));
}
```

**15) Query the output dynamic shape currently used by the RKNN model**

After the rknn_set_input_shape interface is called, pass in the *RKNN_QUERY_CURRENT_OUTPUT_ATTR* command to query the output attribute information currently used by the model. The rknn_tensor_attr structure needs to be created first (Note: RV1106/RV1103 does not support this command).

The sample code is as follows:

```
rknn_tensor_attr cur_output_attrs[io_num.n_output];
memset(cur_output_attrs, 0, io_num.n_output * sizeof(rknn_tensor_attr));
for (uint32_t i = 0; i < io_num.n_output; i++) {
  cur_output_attrs[i].index = i;
  ret = rknn_query(ctx, RKNN_QUERY_CURRENT_OUTPUT_ATTR,
          &(cur_output_attrs[i]), sizeof(rknn_tensor_attr));
}
```

#### 4.4.3.6 rknn_inputs_set

The input data of the model can be set by the rknn_inputs_set function. This function can support multiple inputs, each of which is a *rknn_input* structure object. The user needs to set these object field before passing in *rknn_inputs_set* function (**Note: unavailable on RV1106/RV1103**) .

| | |
|---|---|
| API | rknn_inputs_set |
| Description | Set the model input data. |
| Parameter | *rknn_context context*: The object of rknn_contex. |
| | *uint32_t n_inputs*: Number of inputs. |
| | *rknn_input inputs[]*: Array of rknn_input. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
rknn_input inputs[1];
memset(inputs, 0, sizeof(inputs));
inputs[0].index = 0;
inputs[0].type = RKNN_TENSOR_UINT8;
inputs[0].size = img_width*img_height*img_channels;
inputs[0].fmt = RKNN_TENSOR_NHWC;
inputs[0].buf = in_data;
inputs[0].pass_through = 0;

ret = rknn_inputs_set(ctx, 1, inputs);
```

#### 4.4.3.7 rknn_run

The rknn_run function will perform a model inference. The input data need to be configured by the *rknn_inputs_set* function or zero-copy interface before *rknn_run is* called.

| API | rknn_run |
|---|---|
| Description | Perform a model inference. |
| Parameter | *rknn_context context*: The object of rknn_contex. |
| | *rknn_run_extend* extend*: Reserved for extension. It is not used currently and accepted NULL only. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
ret = rknn_run(ctx, NULL);
```

#### 4.4.3.8 rknn_wait

This interface is used for non-blocking mode inference and **has not been implemented yet**.

#### 4.4.3.9 rknn_outputs_get

The rknn_outputs_get function can get the output data from the model. This function can get multiple output data, each of which is a *rknn_output* structure object that needs to be created and initialized in turn before the function is called (**Note: RV1106/RV1103 unsupported**) .

There are two ways to store buffers for output data:

1) The user allocate and release buffers themselves. In this case, the *rknn_output.is_prealloc* needs to be set to 1, and the *rknn_output.buf* points to users' allocated buffer.

2) The other is allocated by rknn. At this time, the *rknn_output .is_prealloc* needs to be set to 0. After the function is executed, *rknn_output.buf* will be created and store the output data.

| API | rknn_outputs_get |
|---|---|
| Description | Get model inference output data. |
| Parameter | *rknn_context context*: The object of rknn_context. |
| | *uint32_t n_outputs*: Number of output. |
| | *rknn_output outputs[]*: Array of rknn_output. |
| | *rknn_run_extend* extend*: Reserved for extension, currently not used yet. Accepting NULL only. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
rknn_output outputs[io_num.n_output];
memset(outputs, 0, sizeof(outputs));
for (int i = 0; i < io_num.n_output; i++) {
    outputs[i].index = i;
    outputs[i].is_prealloc = 0;
    outputs[i].want_float = 1;
}
ret = rknn_outputs_get(ctx, io_num.n_output, outputs, NULL);
```

#### 4.4.3.10 rknn_outputs_release

The rknn_outputs_release function will release the relevant resources of the *rknn_output* object.

| API | rknn_outputs_release |
|---|---|
| Description | Release the rknn_output object. |
| Parameter | *rknn_context context*: rknn_context object. |
| | *uint32_t n_outputs*: Number of output. |
| | *rknn_output outputs[]*: The array of rknn_output to be released. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
ret = rknn_outputs_release(ctx, io_num.n_output, outputs);
```

### 4.4.3.11 rknn_create_mem_from_mb_blk

**It has not been implemented Currently**.

### 4.4.3.12 rknn_create_mem_from_phys

When the user wants to allocate memory for NPU, the rknn_create_mem_from_phys function can create a *rknn_tensor_mem* structure and return its pointer. This function will pass the physical address, virtual address and size, and the information related to the external memory to the *rknn_tensor_mem* structure.

| API | rknn_create_mem_from_phys |
|---|---|
| Description | Create rknn_tensor_mem structure and allocate memory through physical address. |
| Parameter | *rknn_context context*: rknn_context object. |
| | *uint64_t phys_addr*: The physical address of buffer. |
| | *void *virt_addr:* The virtual address of buffer. |
| | *uint32_t size:* The size of buffer. |
| Return | *rknn_tensor_mem*:* The tensor memory information structure pointer. |

Sample Code:

```
//suppose we have got buffer information as input_phys, input_virt and size
rknn_tensor_mem* input_mems [1];
input_mems[0] = rknn_create_mem_from_phys(ctx, input_phys, input_virt, size);
```

### 4.4.3.13 rknn_create_mem_from_fd

When the user wants to allocate the memory for NPU, the rknn_create_mem_from_fd creates a *rknn_tensor_mem* structure and return its pointer. This function filled with the file descriptor, logical address and size, and the information related to the external memory will be assigned to the *rknn_tensor_mem* structure.

| API | rknn_create_mem_from_fd |
|---|---|
| Description | Create rknn_tensor_mem structure and allocate memory through file descriptor. |
| Parameter | *rknn_context context*: rknn_context object. |
| | *int32_t fd*: The file descriptor of buffer. |
| | *void *virt_addr*: The virtual address of buffer, which indicates the beginning of fd. |
| | *uint32_t size:* The size of buffer. |
| | *int32_t offset:* The offset corresponding for file descriptor and virtual address. |
| Return | rknn_tensor_mem*: The tensor memory information structure pointer. |

Sample Code:

```
//suppose we have got buffer information as input_fd, input_virt and size
rknn_tensor_mem* input_mems [1];
input_mems[0] = rknn_create_mem_from_fd(ctx, input_fd, input_virt, size, 0);
```

#### 4.4.3.14 rknn_create_mem

When the user wants to allocate memory which can be used directly in the NPU, the rknn_create_mem function can create a *rknn_tensor_mem* structure and get its pointer. This function passes in the memory size and initializes the *rknn_tensor_mem* structure at runtime.

| API | rknn_create_mem |
|---|---|
| Description | Create rknn_tensor_mem structure internally and allocate memory during runtime. |
| Parameter | *rknn_context context*: rknn_context object. |
| | *uint32_t size*: The size of buffer. |
| Return | *rknn_tensor_mem**: The tensor memory information structure pointer. |

Sample Code:

```
//suppose we have got buffer size
rknn_tensor_mem* input_mems [1];
input_mems[0] = rknn_create_mem(ctx, size);
```

#### 4.4.3.15 rknn_destroy_mem

The rknn_destroy_mem function destroys the rknn_tensor_mem structure. However, the memory allocated by the user needs to be released manually.

| API | rknn_destroy_mem |
|---|---|
| Description | Destroy rknn_tensor_mem structure. |
| Parameter | *rknn_context context*: rknn_context object. |
| | *rknn_tensor_mem*:* The tensor memory information structure pointer. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
rknn_tensor_mem* input_mems [1];
int ret = rknn_destroy_mem(ctx, input_mems[0]);
```

### 4.4.3.16 rknn_set_weight_mem

If the user has allocated memory for the network weights, after initializing the corresponding *rknn_tensor_mem* structure, the NPU can use the memory through the rknn_set_weight_mem function. This function must be called before calling rknn_run.

| API | rknn_set_weight_mem |
|---|---|
| Description | Set up the rknn_tensor_mem structure containing weights memory information. |
| Parameter | *rknn_context context*: rknn_context object. |
| | *rknn_tensor_mem*:* The tensor memory information structure pointer. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
rknn_tensor_mem* weight_mems [1];
int ret = rknn_set_weight_mem(ctx, weight_mems[0]);
```

### 4.4.3.17 rknn_set_internal_mem

If the user has allocated memory for the internal tensor in network, after initializing the corresponding rknn_tensor_mem structure, the NPU can use the memory through the rknn_set_internal_mem function. This function must be called before calling rknn_run.

| | |
|---|---|
| API | rknn_set_internal_mem |
| Description | Set up the rknn_tensor_mem structure containing internal tensor memory information in network. |
| Parameter | *rknn_context context*: rknn_context object. |
| | *rknn_tensor_mem*:* The pointer to the tensor memory information structure. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
rknn_tensor_mem* internal_tensor_mems [1];
int ret = rknn_set_internal_mem(ctx, internal_tensor_mems[0]);
```

#### 4.4.3.18 rknn_set_io_mem

If the user has allocated memory for the input/output tensor in network, after initializing the corresponding *rknn_tensor_mem* structure, the NPU can use the memory through the rknn_set_io_mem function. This function must be called before calling rknn_run.

| | |
|---|---|
| API | rknn_set_io_mem |
| Description | Set up the rknn_tensor_mem structure containing input/output tensor memory information in network. |
| Parameter | *rknn_context context*: rknn_context object. |
| | *rknn_tensor_mem*:* The pointer to the tensor memory information structure . |
| | rknn_tensor_attr *: The attribute of input or output tensor buffer. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
    rknn_tensor_attr output_attrs[1];
    rknn_tensor_mem* output_mems[1];

    ret = rknn_query(ctx, RKNN_QUERY_NATIVE_OUTPUT_ATTR, &(output_attrs[0]),
sizeof(rknn_tensor_attr));
    output_mems[0] = rknn_create_mem(ctx, output_attrs[0].size_with_stride);
    rknn_set_io_mem(ctx, output_mems[0], &output_attrs[0]);
```

#### 4.4.3.19 rknn_set_input_shape

For dynamic shape input RKNN models, the currently used input shape must be specified before inference. The rknn_tensor_attr* parameters passed in by this interface contain the input shape and corresponding data layout information. Set the dims member of the rknn_tensor_attr structure object to the shape of the input data, and set the fmt member to the corresponding data layout information. Before using this interface, you can first query the input quantity and dynamic shape list supported by the RKNN model through the rknn_query function, and the shape of the input data is required to be in the input shape list supported by the model. Every time you switch to a new input shape, you need to call this interface to set a new shape.

| API | rknn_set_input_shape |
|---|---|
| Description | Set the input shape currently used by the model. |
| Parameter | *rknn_context context*: rknn_context object. |
| | rknn_tensor_attr *: The attribute of the input tensor, including an input shape specified by the user. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
for (int i = 0; i < io_num.n_input; i++) {
    for (int j = 0; j < input_attrs[i].n_dims; ++j) {
        //Read the first dynamic input shape
        input_attrs[i].dims[j] = dyn_range[i].dyn_range[0][j];
    }

    ret = rknn_set_input_shape(ctx, &input_attrs[i]);
    if (ret < 0)
    {
        fprintf(stderr, "rknn_set_input_shape error! ret=%d\n", ret);
        return -1;
    }
}
```

### 4.4.4 RKNN Definition of Data Structcture

#### 4.4.4.1 rknn_sdk_version

The structure *rknn_sdk_version* is used to indicate the version information of the RKNN SDK. The following table shows the definition:

| Field | Type | Meaning |
|---|---|---|
| api_version | char[] | SDK API version information. |
| drv_version | char[] | SDK driver version information. |

#### 4.4.4.2 rknn_input_output_num

The structure *rknn_input_output_num* represents the number of input and output Tensor， The following table shows the definition:

| Field | Type | Meaning |
|---|---|---|
| n_input | uint32_t | The number of input tensor. |
| n_output | uint32_t | The number of output tensor. |

#### 4.4.4.3 rknn_input_range

The structure rknn_input_range represents an input support shape list information. It contains the input index, the number of supported shapes, data layout format, name and shape list. The definition of the specific structure is shown in the following table:

| Field | Type | Meaning |
|---|---|---|
| index | uint32_t | The index position of the input. |
| shape_number | uint32_t | The number of input shapes supported by the RKNN model. |
| fmt | rknn_tensor_format | The data layout format corresponding to the shape. |
| name | char[] | The name of the input. |
| dyn_range | uint32_t[][] | The input shape list, which is a two-dimensional array containing multiple shape arrays, and the shape is stored first. |
| n_dims | uint32_t | The number of valid dimensions for each shape array. |

#### 4.4.4.4 rknn_tensor_attr

The structure *rknn_tensor_attr* represents the attribute of the model's Tensor. The following table shows the definition:

| Field | Type | Meaning |
|---|---|---|
| index | uint32_t | Indicates the index position of the input and output Tensor. |
| n_dims | uint32_t | The number of Tensor dimensions. |
| dims | uint32_t[] | Values for each dimension. |
| name | char[] | Tensor name. |
| n_elems | uint32_t | The number of Tensor data elements. |
| size | uint32_t | The memory size of Tensor data. |
| fmt | rknn_tensor_format | The format of Tensor dimension, has the following format:<br>**RKNN_TENSOR_NCHW**<br>**RKNN_TENSOR_NHWC**<br>**RKNN_TENSOR_NC1HWC2** |
| type | rknn_tensor_type | Tensor data type, has the following data types:<br>**RKNN_TENSOR_FLOAT32**<br>**RKNN_TENSOR_FLOAT16**<br>**RKNN_TENSOR_INT8**<br>**RKNN_TENSOR_UINT8**<br>**RKNN_TENSOR_INT16**<br>**RKNN_TENSOR_UINT16**<br>**RKNN_TENSOR_INT32**<br>**RKNN_TENSOR_INT64**<br>**RKNN_TENSOR_BOOL** |
| qnt_type | rknn_tensor_qnt_type | Tensor Quantization Type, has the following types of quantization:<br>**RKNN_TENSOR_QNT_NONE:** Not quantized;<br>**RKNN_TENSOR_QNT_DFP**: Dynamic fixed point quantization;<br>**RKNN_TENSOR_QNT_AFFINE_ASYMMETRIC**: Asymmetric quantification. |
| fl | int8_t | RKNN_TENSOR_QNT_DFP: quantization parameter. |
| scale | float | RKNN_TENSOR_QNT_AFFINE_ASYMMETRIC: quantization parameter. |

| | | |
|---|---|---|
| w_stride | uint32_t | The number of pixels that actually store one line of image data, which is equal to the number of valid data pixels in one row + the number of invalid pixels that are filled in for the hardware to quickly jump to the next line (unit: per pixel) . |
| size_with_stride | uint32_t | The actual byte size of image data (including the byte size of filled invalid pixels). |
| pass_through | uint8_t | 0: means unconverted data. 1: means converted data, Note: Conversion includes normalization and quantization. |
| h_stride | uint32_t | This is only used in the context of multi-batch input and can be set by users. The purpose of this is to allow NPU to read out beginning of memory address for every batch correctly. It is equivalent of original model input height + the number of invalid pixels that are filled in for the hardware to quickly jump to the next line . If its value is 0, it is the same as model input height (unit: per pixel) . |

#### 4.4.4.5 rknn_perf_detail

The structure rknn_perf_detail represents the performance details of the model. The definition of the structure is shown in the following table: (**Note: RV1106/RV1103 unsupported**)

| Field | Type | Meaning |
|---|---|---|
| perf_data | char* | The performance details contain the running time of each layer of the network stored in string type. |
| data_len | uint64_t | The data length of string type of perf_data. |

### 4.4.4.6 rknn_perf_run

The structure rknn_perf_run represents the total inference time of the model. The definition of the structure is shown in the following table: (**Note: RV1106/RV1103 unsupported**)

| Field | Type | Meaning |
|---|---|---|
| run_duration | int64_t | The total inference time of the network (not including setting input/output) in microseconds. |

### 4.4.4.7 rknn_mem_size

The structure *rknn_mem_size* represents the memory allocation when the model is initialized. The definition of the structure is shown in the following table:

| Field | Type | Meaning |
|---|---|---|
| total_weight_size | uint32_t | The memory size allocated for weights of the network. |
| total_internal_size | uint32_t | The memory size allocated for internal tensor of the network. |
| total_dma_allocated_size | uint64_t | All dma memory size allocated for the network. |
| total_sram_size | uint32_t | Only for RK3588, memory size of SRAM reserved for NPU (Referring to the <<RK3588_NPU_SRAM_usage.md>> for more details) . |
| free_sram_size | uint32_t | Only for RK3588, the current available SRAM (Referring to the <<RK3588_NPU_SRAM_usage.md>> for more details) . |
| reserved[12] | uint32_t | Reserve. |

#### 4.4.4.8 rknn_tensor_mem

The structure *rknn_tensor_mem* represents the tensor memory information. The definition of the structure is shown in the following table:

| Field | Type | Meaning |
|---|---|---|
| virt_addr | void* | The virtual address of tensor. |
| phys_addr | uint64_t | The physical address of tensor. |
| fd | int32_t | The file descriptor of tensor. |
| offset | int32_t | The offset of fd and virtual address. |
| size | uint32_t | The actual size of tensor. |
| flags | uint32_t | rknn_tensor_mem has the following type of flags:<br>**RKNN_TENSOR_MEMORY_FALGS_ALLOC_INSIDE:** indicates that the rknn_tensor_mem structure is created during runtime;<br>**RKNN_TENSOR_MEMORY_FLAGS_FROM_FD:** It represents the rknn_tensor_mem created by fd;<br>**RKNN_TENSOR_MEMORY_FLAGS_FROM_PHYS:** It represents rknn_tensor_mem created by physical address;<br>The user does not need to pay attention to the flags. |
| priv_data | void* | private data. |

#### 4.4.4.9 rknn_input

The structure *rknn_input* represents a data input to the model, used as a parameter to the rknn_inputs_set function. The following table shows the definition:

| Field | Type | Meaning |
|---|---|---|
| index | uint32_t | The index position of this input. |
| buf | void* | The pointer of the input data buffer. |
| size | uint32_t | The memory size of the input data buffer. |
| pass_through | uint8_t | When set to 1, buf will be directly set to the input node of the model without any pre-processing. |
| type | rknn_tensor_type | The type of input data. |
| fmt | rknn_tensor_format | The format of input data. |

#### 4.4.4.10 rknn_output

The structure *rknn_output* represents a data output of the model, used as a parameter to the rknn_outputs_get function. This structure will be assigned with data after calling rknn_outputs_get. The following table shows the definition of each member of *rknn_output*:

| Field | Type | Meaning |
|---|---|---|
| want_float | uint8_t | Indicates whether the output data needs to be converted to float type. |
| is_prealloc | uint8_t | Indicates whether the buffer that stores the output data is pre-allocated. |
| index | uint32_t | The index position of this output. |
| buf | void* | The pointer pointing to the output data buffer. |
| size | uint32_t | Output data buffer size in byte. |

#### 4.4.4.11 rknn_init_extend

The structure *rknn_init_extend* represents the extended information when the model is initialized. **It is not available currently on RV1106/RV1103**. The definition of the structure is shown in the following table:

| Field | Type | Meaning |
|---|---|---|
| ctx | rknn_context | The initialized rknn_context object. |
| real_model_offset | int32_t | The real offset that is in .rknn model file. Only valid when using file path as initialized parameter. |
| real_model_size | uint32_t | The real size of rknn model file. Only valid when using file path as initialized parameter. |
| reserved | uint8_t[120] | The reserved data. |

#### 4.4.4.12 rknn_run_extend

The structure *rknn_run_extend* represents the extended information during model inference. **It is not available currently**. The definition of the structure is shown in the following table:

| Field | Type | Meaning |
|---|---|---|
| frame_id | uint64_t | The index of frame of current inference. |
| non_block | int32_t | 0 means blocking mode, 1 means non-blocking mode, non-blocking mode means that the rknn_run call returns immediately. |
| timeout_ms | int32_t | The timeout of inference in milliseconds. |
| fence_fd | int32_t | For the non-blocking inference (**Not Available**) . |

#### 4.4.4.13 rknn_output_extend

The structure *rknn_output_extend* means to obtain the extended information of the output. **It is not available currently**. The definition of the structure is shown in the following table:

| Field | Type | Meaning |
|---|---|---|
| frame_id | int32_t | The frame index of the output result. |

#### 4.4.4.14 rknn_custom_string

The structure rknn_custom_string represents the custom string set by the user when converting the RKNN model. The definition of the structure is shown in the following table:

| Field | Type | Meaning |
|---|---|---|
| string | char[] | User-defined string. |

### 4.4.5 Instruction of API Input and Output

#### 4.4.5.1 General Input and Output of API (Non Zero Copy)

**Note: RV1106/RV1103 unsupported.**

**1) rknn_inputs_set**

When calling this API, it can set up the parameter from rknn_inputs_set, that is, the input data type and size can be configured by the rknn_input input[] .

The detail explanation of input: The pass_through is one of the member of rknn_input structure. If pass_through is set as 1, then the input data will be imported and computed inside the model in which none of the conversion will happen. If it is 0, the input tensor will be conveted first by the rknn api based on the fmt and type from the rknn_input struct, which is happened inside the RKNN API and fed into model.

When using this rknn_input_set, user should query **RKNN_QUERY_NATIVE_INPUT_ATTR** rather than **RKNN_QUERY_INPUT_ATTR.** Depending on the fmt or laypout, the preprocessing is different correspondingly.

**a) When layout is RKNN_TENSOR_NCHW:** the input usually is 4 dimensional and data type is bool or int64, when passing to NPU, the data layout need to be arranged as NCHW for NPU.

**b) When layout is RKNN_TENSOR_NHWC:** the input usually is 4 dimensional and data type is one of the float32/float16/int8/uint8. Also, the number of input channel is either 1, 3 or 4. When passing to NPU, data layout need to be arranged into NHWC layout. It is worth nothing that when setting pass_through is 1, the width may have an alignment with stride depending on the w_stride from rknn_input_attr.

**c) When layout is RKNN_TENSOR_NC1HWC2:** the input usually is 4 dimensional and data type is float16/int8. Also, the input channel is not 1, 3 and 4. When pass_through is 0, the input data layout need to be arranged as NHWC. **When pass_through is 1, the input data layout need to reshaped as RKNN_TENSOR_NC1HWC2.**

**d) When layout is RKNN_TENSOR_UNDEFINED:** Usually, the input is not 4 dimensional. When passing to NPU, the input layout need to arranged according to the input model layout in ONNX!. In this situation, the NPU will not do anything related to process of mean/std and layout conversion.

The relationship between input data type and data type queried from **RKNN_QUERY_INPUT_ATTR** is shown in table below.

| **Input data type from RKNN_QUERY_INPUT_ATTR** | **Available Input data type** |
|---|---|
| bool | bool |
| int8 | int8 / uint8 / float32 |
| float16 | uint8 / float16 / float32 |
| int64 | int64 |

Table4-1: The available input data type and model input data type

**2) rknn_outputs_get**

When calling this function, it can set up want_float, which is one of the member of rknn_output structure,  to select the input data type, layout, etc. When the rknn-toolkit 2 tool before version 1.2.0 converts the RKNN model, the data type of the output layer is fixed to float32 by default, so even if it is want_ Float = 0, the result of non quantitative model is still float32. In 1.2.0, including the RKNN model converted by the toolkit after 1.2.0, the output of int8 model is int8 data type; In particular, if the last layer of int8 model is float16 output, the model output is float16 data type, for example, the last layer is softmax layer; The float16 model is output as float16 data type.

Table4-2: RK3562/RK3566/RK3568/RK3588 supports input configuration without zero copy

| want_float | Output data Type | Supported output layout | Remark |
|---|---|---|---|
| 1 | float32 | NCHW | |
| 0 | int8 | NCHW | Only support i8 model |
| | float16 | NCHW | |

#### 4.4.5.2 Input and Output with Zero Copy of API

Two functions need to use for this purpose:

**rknn_create_mem**,

**rknn_set_io_mem**

The specification of input data (involving the size, data type, layout, etc.) should be configured according to the actual data that need to be allocated by the zero copy memory.

For RK3562, RK3566/RK3568 and RK3588, NPU supports the same input configuration the mentioned in the section of **General Input and Output of API .**

For RV1106 and RV1103, NPU supports the input configuration as follow:

Table4-3: RV1106/RV1103 supports input configuration with zero copy

| Data Type of Input | pass_through | Number of Channels | Supported input layout | Remark |
|---|---|---|---|---|
| uint8_t | 0 | - | NHWC | Only support i8 model |
| int8 | 1 | 1,3,4 | NHWC | |
| int8 | 1 | Not 1,3,4 channel | NC1HWC2 | Only support i8 model |

For the output data, when not in the mode of NATIVE_LAYOUT, it is suggested considering the actual data inside the zero copy memory to configure the corresponding size, data type and data layout for output. When in the NATIVE_LAYOUT mode, it is recommended utilizing the default configuration that is from the one queried from the RKNN_QUERY_NATIVE_INPUT_ATTR and data size should adopt to size_with_stride. The specification of output is shown below.

| Output Data Type | Available Output Layout | Remark |
|---|---|---|
| float32 | NCHW | |
| int8 | NCHW | Only support i8 model |
| int8 | NC1HWC2 | Only support i8 model |
| float16 | NCHW | Only support fp16 model |
| float16 | NC1HWC2 | Only support fp16 model |

Table4-4: RK3562/RK3566/RK3568/RK3588 supports output setting. (With zero copy interface)

| Output Data Type | Available Output Layout | Remark |
|---|---|---|
| float32 | NHWC | **Unsupported currently** |
| int8 | NHWC | **Unsupported currently** |
| int8 | NC1HWC2 | Only support i8 model |

Table4-5: RV1106/RV1103 supports output setting. (With zero copy interface)

#### 4.4.5.3 Instruction of looking up NATIVE_LAYOUT parameter

There are two parameters listed below.

**RKNN_QUERY_NATIVE_INPUT_ATTR**

**RKNN_QUERY_NATIVE_OUTPUT_ATTR**

Those two can query the attribute of input and output tensor. In most of the time, this native attribute has the best performance. When querying this attribute, the rknn_tensor_format might be the NC1HWC2, which is a special memory arrangement. The memory address from low to high is C2->W->H->C1->N, among which the C2 has the fastest changing rate and N is the slowest one. The value of C2 may vary depending on the platform and data type. The specification of C2 is shown in the table below.

Table4-6: The value of C2 under different platform

| platform | int8(quantization) | float16(non-quantization) | Int16(quantization) |
|---|---|---|---|
| RK3566/RK3568 | 8 | 4 | 4 |
| RK3588 | 16 | 8 | 8 |
| RV1106/RV1103 | 16 | 8 | 8 |
| RK3562 | 16 | 8 | 8 |

For example, assuming on the RK3566/RK3568, there is an output tensor whose memory layout is 1x1x1x32 (NHWC) from the orignal model. Then, the original model is converted to RKNN model. When querying the output tensor from this new model with the native attribute, the memory layout of this output tensor will be converted to 1x4x1x1x8 (NC1HWC2), where C1 = [32/8] =4, C2 = 8, the [] denotes the rounding up to cloest integer. Take the NC1HWC2 arrangement of int8 data of RK3566/RK3568 in memory as an example, where C2 = 8, as shown in Figure 4-9.

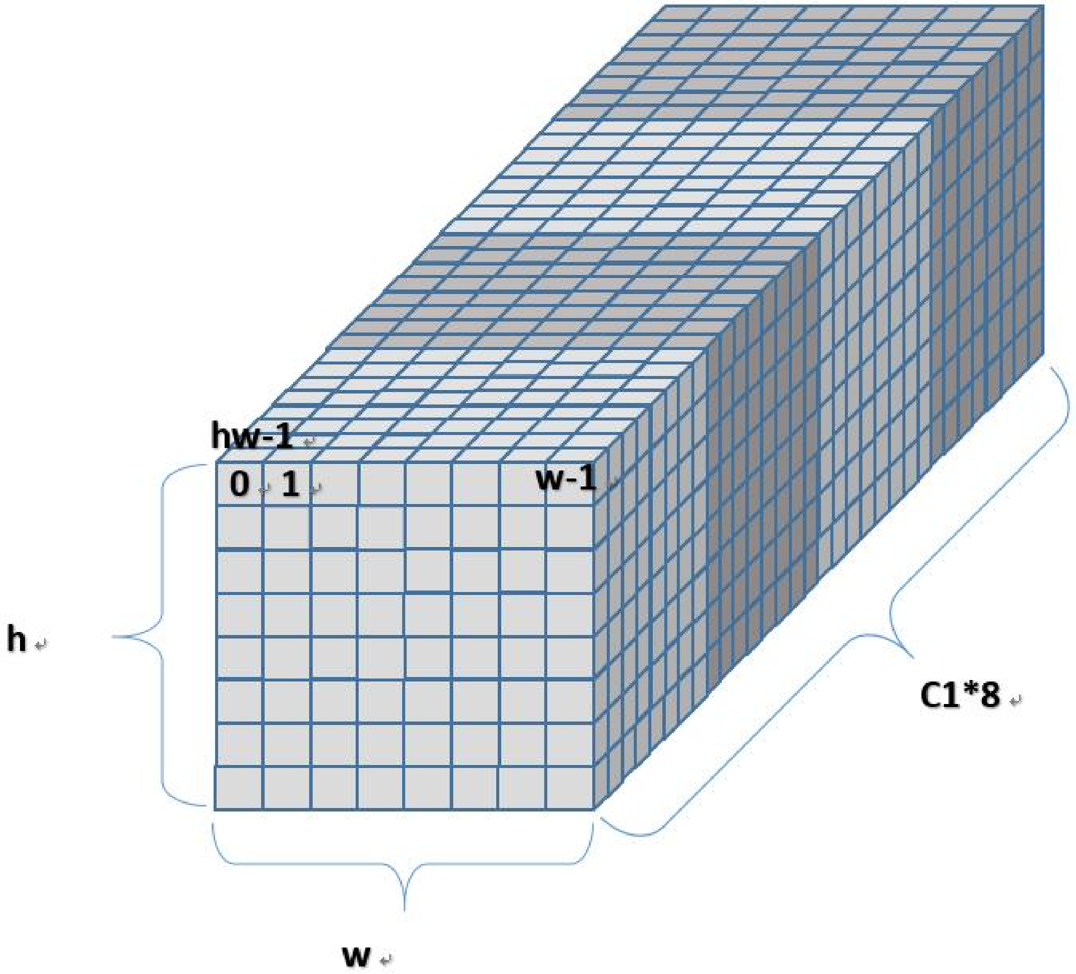

Fig4-9 The arrangement of int8 data of RK3566/RK3568 in NC1HWC2 in memory

（1）NC1HWC2 to NCHW: NC1HWC2 arranged in int8 data is converted to NCHW arranged in int8 data, as shown below.

```
/*
 *src:   NC1HWC2 input tensor addr
 *dst:    NCHW output tensor addr
 *dims:  input  NC1HWC2 shape
 *channel: output NCHW channel value
 * h:       output NCHW h value
 * w:       output NCHW w value
 */
int NC1HWC2_to_NCHW(const int8_t* src, float* dst, int* dims, int channel, int h, int w)
{
  int batch  = dims[0];
  int C1     = dims[1];
  int C2     = dims[4];
  int hw_src = dims[2] * dims[3];
  int hw_dst = h * w;
  for (int i = 0; i < batch; i++) {
   src = src + i * C1 * hw_src * C2;
   dst = dst + i * channel * hw_dst;
   for (int c = 0; c < channel; ++c) {
     int          plane  = c / C2;
     const int8_t* src_c  = plane * hw_src * C2 + src;
     int          offset = c % C2;
     for (int cur_h = 0; cur_h < h; ++cur_h)
       for (int cur_w = 0; cur_w < w; ++cur_w) {
         int cur_hw                      = cur_h * w + cur_w;
         dst[c * hw_dst + cur_h * w + cur_w] = src_c[C2 * cur_hw + offset];
       }
   }
  }

  return 0;
}
```

（2）NC1HWC2 to NHWC: NC1HWC2 arranged in int8 data is converted to NHWC arranged in int8 data, as shown below.

```
/*
 *src:   NC1HWC2 input tensor addr
 *dst:    NCHW output tensor addr
 *dims:   input  NC1HWC2 shape
 *channel: output NCHW channel value
 * h:       output NCHW h value
 * w:       output NCHW w value
 */
int NC1HWC2_to_NHWC(const int8_t* src, int8_t* dst, int* dims, int channel, int h, int w)
{
  int batch  = dims[0];
  int C1     = dims[1];
  int C2     = dims[4];
  int hw_src = dims[2] * dims[3];
  int hw_dst = h * w;
  for (int i = 0; i < batch; i++) {
    src = src + i * C1 * hw_src * C2;
    dst = dst + i * channel * hw_dst;
    for (int cur_h = 0; cur_h < h; ++cur_h) {
      for (int cur_w = 0; cur_w < w; ++cur_w) {
        int cur_hw = cur_h * dims[3] + cur_w;
        for (int c = 0; c < channel; ++c) {
          int        plane = c / C2;
          const auto* src_c = plane * hw_src * C2 + src;
          int        offset = c % C2;
          dst[cur_h * w * channel + cur_w * channel + c] = src_c[C2 * cur_hw + offset];
        }
      }
    }
  }  return 0;
}
```

After calling the rknn_query with **RKNN_QUERY_NATIVE_INPUT_ATTR** or **RKNN_QUERY_NATIVE_OUTPUT_ATTR**, this API will return the configuration with the best performance to the **rknn_tensor_attr**. If the user has got different input or output data configuration differed from what they desire, they can customize the configuration of the query part. However, it must be consistent with the table-**3** and table-**4** shown above. For example, if the queried data type is **uint8** and users want to configure this output data type as **float32**, users should multiply the size in **rknn_tensor_attr** structure by 4 since the float32 is 4 bytes long. Also, the data type in rknn_tensor_attr should be changed to **RKNN_TENSOR_FLOAT32**. With regrading to querying in the native mode using the above two parameters, users should use the zero copy interface to fetch the result of input and output.

## 4.5 Matrix Multiplication API

The matrix multiplication interface is a set of APIs to efficiently perform matrix multiplication operations on the NPU (It is currently a beta version, and RV1103/RV1106 is not currently supported). Multiple data types and memory arrangements are supported. This API can accept two matrices as input and return their product matrix. The operation is defined as follows:

$$C=A*B$$

here:

A, B and C are 2-dimensional matrices,

A is a matrix with M rows and K columns,

B is a matrix with K rows and N columns,

C is a matrix with M rows and N columns.

### 4.5.1 Matrix Multiplication API Reference

#### 4.5.1.1 rknn_matmul_create

This function creates a matrix multiplication operation handle and completes the initialization of the matrix multiplication context for performing matrix multiplication operations. At the same time, this function will read the matrix multiplication specification information and get the input and output tensor attributes. Here, the rknn_matmul_ctx handle and rknn_context are the same data structure.

| API | rknn_matmul_create |
|---|---|
| Description | Initializes the matrix multiplication context. |
| Parameters | rknn_matmul_ctx* ctx: Matrix multiplication context handle. |
| | rknn_matmul_info* info: Pointer to the specification information structure of matrix multiplication. |
| | rknn_matmul_io_attr* io_attr: Pointer to the matrix multiplication input and output tensor attribute structure. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
rknn_matmul_info info;
memset(&info, 0, sizeof(rknn_matmul_info));
info.M          = 4;
info.K          = 64;
info.N          = 32;
info.type         = RKNN_TENSOR_INT8;
info.native_layout  = 0;
info.perf_layout    = 0;

rknn_matmul_io_attr io_attr;
memset(&io_attr, 0, sizeof(rknn_matmul_io_attr));

int ret = rknn_matmul_create(&ctx, &info, &io_attr);
if (ret < 0) {
  printf("rknn_matmul_create fail! ret=%d\n", ret);
  return -1;
}
```

#### 4.5.1.2 rknn_matmul_set_io_mem

This function is used to set the input/output memory for the matrix multiplication operation. Before calling this function, first use the rknn_tensor_mem structure pointer created by the rknn_create_mem interface, and then pass it into the function with the rknn_matmul_tensor_attr structure pointer of the matrix A, B or C returned by the rknn_matmul_create function, and set the input and output memory to the matrix multiplication in context. Before calling this function, the data of matrix A and matrix B should be prepared according to the memory arrangement configured in rknn_matmul_info.

| API | rknn_matmul_set_io_mem |
|---|---|
| Description | Sets the input/output memory for matrix multiplication. |
| Parameters | rknn_matmul_ctx* ctx: Matrix multiplication context handle. |
| | rknn_tensor_mem* mem: Pointer to tensor memory information structure. |
| | rknn_matmul_tensor_attr* attr: Pointer to matrix multiplication input and output tensor attribute structure. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
// Create A
rknn_tensor_mem* A = rknn_create_mem(ctx, io_attr.A.size);
if (A == NULL) {
  printf("rknn_create_mem fail!\n");
  return -1;
}
memset(A->virt_addr, 1, A->size);
rknn_matmul_io_attr io_attr;
memset(&io_attr, 0, sizeof(rknn_matmul_io_attr));

int ret = rknn_matmul_create(&ctx, &info, &io_attr);
if (ret < 0) {
  printf("rknn_matmul_create fail! ret=%d\n", ret);
  return -1;
}
// Set A
ret = rknn_matmul_set_io_mem(ctx, A, &io_attr.A);
if (ret < 0) {
  printf("rknn_matmul_set_io_mem fail! ret=%d\n", ret);
  return -1;
}
```

#### 4.5.1.3 rknn_matmul_set_core_mask

This function is used to set the available NPU cores for matrix multiplication (only RK3588 is supported). Before calling this function, you need to initialize the matrix multiplication context through the rknn_matmul_create function. The mask value that can be set by this function specifies the cores that need to be used to improve the performance and efficiency of matrix multiplication operations.

| | |
|---|---|
| API | rknn_matmul_set_core_mask |
| Description | Sets the NPU core mask for matrix multiply operations. |
| Parameters | rknn_matmul_ctx* ctx: Matrix multiplication context handle. |
| | rknn_core_mask core_mask: The NPU core mask value of the matrix multiplication operation, which is used to specify the available NPU cores. Each bit of the mask represents a core. If the corresponding bit is 1, it means that the core is available; otherwise, it means that the core is unavailable (see rknn_set_core_mask API parameters for detailed mask description). |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
rknn_matmul_set_core_mask(ctx, RKNN_NPU_CORE_AUTO);
```

#### 4.5.1.4 rknn_matmul_run

This function is used to run a matrix multiplication operation and save the result in the output matrix C. Before calling this function, the input matrices A and B need to prepare data first, and set them to the input buffer through the rknn_matmul_set_io_mem function. The output matrix C needs to be set to the output buffer through the rknn_matmul_set_io_mem function, and the tensor attribute of the output matrix is obtained through the rknn_matmul_create function.

| | |
|---|---|
| API | rknn_matmul_run |
| Description | Perform a matrix multiplication operation. |
| Parameters | rknn_matmul_ctx* ctx: Matrix multiplication context handle. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
int ret = rknn_matmul_run(ctx);
```

#### 4.5.1.5 rknn_matmul_destroy

This function is used to destroy the matrix multiplication operation context and release related resources. After using the matrix multiplication handle created by the rknn_matmul_create function, you need to call this function to destroy it.

| | |
|---|---|
| API | rknn_matmul_destroy |
| Description | Destroys the matrix multiply operation context. |
| Parameters | rknn_matmul_ctx* ctx: Matrix multiplication context handle. |
| Return | int: Error code (See RKNN Error Code). |

Sample Code:

```
int ret = rknn_matmul_destroy(ctx);
```

### 4.5.2 Definition of matrix multiplication data structure

#### 4.5.2.1 rknn_matmul_info

rknn_matmul_info indicates the specification information for performing matrix multiplication, which includes the size of matrix multiplication, data type and memory arrangement of input and output matrices. The definition of the structure is shown in the following table:

| Field | Type | Meaning |
|---|---|---|
| M | int32_t | The number of rows of the A matrix. |
| K | int32_t | The number of rows of the A matrix (the number of columns of the B matrix). |
| N | int32_t | The number of rows of the C matrix. |
| type | rknn_tensor_type | The data type of the input matrix. |
| native_layout | int32_t | The memory layout of the B matrix.<br>0: Indicates [K,N] shape arrangement.<br>1: Indicates [N1, K1, N2, K2] shape arrangement. |
| perf_layout | int32_t | Memory layout of matrix A and matrix C.<br>0: Indicates that matrix A is arranged in [M, K] shape, and matrix C is arranged in [M, N] shape.<br>1: Indicates that matrix A is arranged in the shape of [K1, M, K2], and matrix C is arranged in the shape of [N1, M, N2]. |

#### 4.5.2.2 rknn_matmul_tensor_attr

rknn_matmul_tensor_attr represents the attribute of each matrix tensor, which includes the name, shape, size and data type of the matrix. The definition of the structure is shown in the following table:

| Field | Type | Meaning |
|---|---|---|
| name | char[] | The name of the matrix. |
| n_dims | uint32_t | The number of dimensions of the matrix. |
| dims | uint32_t[] | The shape of the matrix. |
| size | uint32_t | the size of the matrix. |
| type | rknn_tensor_type | The data type of matrix |

#### 4.5.2.3 rknn_matmul_io_attr

rknn_matmul_io_attr represents the attributes of all input and output tensors of the matrix, which includes the attributes of matrices A, B, and C. The definition of the structure is shown in the following table:

| Field | Type | Meaning |
|---|---|---|
| A | rknn_matmul_tensor_attr | The tensor attribute of matrix A. |
| B | rknn_matmul_tensor_attr | The tensor attribute of matrix B. |
| C | rknn_matmul_tensor_attr | The tensor attribute of matrix C. |

### 4.5.3 Matrix multiplication input/output memory layout and data type description

When setting the memory of the input matrix A and B and the output matrix C, the data of the input matrix A and B need to be prepared according to a specific arrangement, and the data of the output matrix C is also read according to a specific arrangement.

For the input matrix A, suppose the number of rows is M, the number of columns is K, and the memory layout is as follows:

1. If perf_layout=0, the shape of the input matrix A is [M,K], the length of the array is M×K, and all elements of A are stored.

2. If perf_layout=1, the shape of the input matrix A is [K1, M, K2], and the values of K1, M, and K2 under different chip platforms and data types are shown in Table 4-7 (the division results in the table is rounded up and the extra parts are filled with 0):

| Platform | int8 | float16 |
|---|---|---|
| RK3566/RK3568 | [K/8,M,8] | [K/4,M,4] |
| RK3588 | [K/16,M,16] | [K/16,M,16] |
| RK3562 | [K/16,M,16] | [K/16,M,16] |

Table 4-7 Data type and shape of input matrix A under different chip platforms and data types

For the input matrix B, assuming that the number of rows is K and the number of columns is N, the memory layout is as follows:

1. If native_layout=0, the shape of the input matrix B is [K,N], the length of the array is K×N, and

all elements of B are stored.

2. If native_layout=1, the shape of the input matrix B is [N1, K1, N2, K2], and the values of N1, K1, N2, and K2 under different chip platforms and data types are shown in Table 4-8 (the division results in the table is rounded up, and the extra parts are filled with 0):

| Platform | int8 | float16 |
|---|---|---|
| RK3566/RK3568 | [N/16,K/32,16,32] | [N/8,K/16,8,16] |
| RK3588 | [N/32,K/32,32,32] | [N/16,K/32,16,32] |
| RK3562 | [N/16,K/32,16,32] | [N/8,K/16,8,16] |

Table 4-8 Data type and shape of input matrix B under different chip platforms and data types

For the input matrix C, assuming that the number of rows is M and the number of columns is N, the memory layout is as follows:

1. If perf_layout=0, the shape of the input matrix C is [M,N], the length of the array is M×N, and all elements of C are stored.

2. If perf_layout=1, the shape of the input matrix A is [N1, M, N2], and the values of N1, M, and N2 under different chip platforms and data types are shown in Table 4-9 (the division results in the table is rounded up and the extra parts are filled with 0):

| Platform | int32 | float32 |
|---|---|---|
| RK3566/RK3568 | [K/4,M,4] | [K/4,M,4] |
| RK3588 | [K/4,M,4] | [K/4,M,4] |
| RK3562 | [K/4,M,4] | [K/4,M,4] |

Table 4-9 Data type and shape of input matrix C under different chip platforms and data types

### 4.5.4 Matrix Multiplication Specification Limits

The matrix multiplication operator library is implemented based on the NPU hardware architecture and is limited by hardware specifications. The parameters are restricted as follows:

#### 4.5.4.1 Shape constraints

First, for an int8 input matrix, K is required to be less than or equal to 4096; for a float16 input matrix, K is required to be less than or equal to 2048. Secondly, both K and N can only be greater than 32 and be multiples of 32.

#### 4.5.4.2 Input data type restrictions

The input only supports 8-bit signed integer and 16-bit floating point (float16), and the output only supports 32-bit signed integer and 32-bit floating point.

## 4.6 RKNN Error Code

The return code of the RKNN API function is defined as shown in the following table.

| Error Code | Message |
|---|---|
| RKNN_SUCC（0） | Execution is successful. |
| RKNN_ERR_FAIL (-1) | Execution error. |
| RKNN_ERR_TIMEOUT (-2) | Execution timeout. |
| RKNN_ERR_DEVICE_UNAVAILABLE (-3) | NPU device is unavailable. |
| RKNN_ERR_MALLOC_FAIL (-4) | Memory allocation is failed. |
| RKNN_ERR_PARAM_INVALID (-5) | Parameter error. |
| RKNN_ERR_MODEL_INVALID (-6) | RKNN model is invalid. |
| RKNN_ERR_CTX_INVALID (-7) | rknn_context is invalid. |
| RKNN_ERR_INPUT_INVALID (-8) | rknn_input object is invalid. |
| RKNN_ERR_OUTPUT_INVALID (-9) | rknn_output object is invalid. |
| RKNN_ERR_DEVICE_UNMATCH (-10) | Version does not match. |
| RKNN_ERR_INCOMPATILE_OPTIMIZATION_LEVEL_VERSION (-12) | This RKNN model use optimization level mode, but not compatible with current driver. |
| RKNN_ERR_TARGET_PLATFORM_UNMATCH (-13) | This RKNN model doesn’t compatible with current platform. |

## 4.7 NPU SDK Instruction

### 4.7.1 SDK Content Explain

The NPU SDK for RK3562 RK3566/RK3568 and RK3588 contains examples for the API usage, including the NPU library, server programs, doc. The server program is called rknn_server, which resides on the development board. With the communication set up by the server program, The developer should connect the development board with the PC via the USB and use the rknn-toolkit2 API with python interface to run the model. The detail instruction of how to install is written in <<rknn_server_proxy.md>>.

The overall structure of the content is shown below,

```
├── doc
├── examples                            # Example Program
│   ├── 3rdparty
│   ├── librknn_api_android_demo
│   ├── README.md
│   ├── rknn_api_demo
│   ├── rknn_benchmark
│   ├── rknn_common_test
│   ├── rknn_dynamic_shape_input_demo       #Dynamic Shape Input Demo
│   ├── rknn_internal_mem_reuse_demo
│   ├── rknn_mobilenet_demo
│   ├── rknn_multiple_input_demo
│   ├── rknn_ssd_demo
│   ├── rknn_yolov5_android_apk_demo
│   ├── rknn_yolov5_demo
│   └── RV1106_RV1103
├── README.md
├── rknn_server_proxy.md                # The instruction of server program (used for
│                                       inference with board)
└── runtime
    ├── RK356X                          # RK3566/RK3568/RK3562 library and content
    │   │                               # of server program on the board side
    │   ├── Android
    │   │   ├── librknn_api
    │   │   └── rknn_server
    │   └── Linux
    │       ├── librknn_api
    │       └── rknn_server
    └── RK3588                          # RK3588  library and content of server program
    │   │                               # on the board side
    │   ├── Android
    │   │   ├── librknn_api
    │   │   └── rknn_server
    │   └── Linux
    │       ├── librknn_api
    │       └── rknn_server
    └─ RV1106                           # RV1106/RV1103 library and content of server
                                        # program on the board side
          └── Linux
              └── librknn_api
              └── rknn_server
```

# 4.8 Debugging

## 4.8.1 Log Level

The library for NPU will rely on the environment variable on the board to generate a set of debugging logs or files that are suitable for developers to debug the program. The command for setting up debugging log is shown below:

```
export  RKNN_LOG_LEVEL=<level_number>
```

where <level_number> represents the category of debugging log, ranging from 0 to 5. The detail of each type of log is listed below.

Table-4-7: The output log of each category

| Level | Output Log. |
|---|---|
| 0 | Printing error message only. |
| 1 | Printing both error and warning log. |
| 2 | Printing both hints and log in level 1. |
| 3 | Printing both debugging log and log in level 2 . |
| 4 | Printing both log in level 3 and information in each layer, influencing the performance of rknn_run when enabling this log. |
| 5 | Printing both message in level 4 and output data in intermediate layer, influencing the performance of rknn_run when enabling this log. |

As an example, for querying the performance of each layer, the developer can set the following environment variable.

```
export RKNN_LOG_LEVEL=4
```

When completing the debugging, to reset the debugging log to level 0, using the following commands:

```
unset RKNN_LOG_LEVEL
```

### 4.8.2 Printing for each layer

The following setting will open the performance analysis for each layer:

RKNN_DUMP_QUANT:

When the value of this is 0: this will dump float32 in npy format by default. If this value was 1, the output type will be the same type with the model. For instance, the int8 model would have the int8 output type for tensors dump from each layer when this value was 1.

RKNN_DUIMP_DIR:

This will set up the path for saving these output tensors dump from each layer.

RKNN_LOG_LEVEL:

When this value is set as 5, the output tensor will be dumped from each layer saved in npy format.

When this value is set as 6, the output tensor will be saved as format in both npy and txt in NC1HWC2 layout. For example, for saving results dump from each layer by setting these environment variables as shown below.

```
export RKNN_DUMP_QUANT=0
export RKNN_DUMP_DIR =/data/dumps
export RKNN_LOG_LEVEL =5
```

### 4.8.3 Profiling

#### 4.8.3.1 Checking the environment on the development board

Generally, the frequency of each hardware unit on the development board is not fixed, resulting in the fluctuation and uncertainty on the performance of module running on the board. To avoid this, the developer should fix the frequency of each hardware unit before testing modules for assessment.

1. **Fix Frequency for CPU:**

**(1) Query the frequency of CPU:**

```
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq
```

**(2) Fix the frequency of CPU:**

```
# Query avilable frequency of CPU(different platforms will vary)
cat /sys/devices/system/cpu/cpufreq/policy0/scaling_available_frequencies
408000 600000 816000 1008000 1200000 1416000 1608000 1704000
# Fix the CPU frequency, e.g., fix at 1.7GHz
echo userspace > /sys/devices/system/cpu/cpufreq/policy0/scaling_governor
echo 1704000 > /sys/devices/system/cpu/cpufreq/policy0/scaling_setspeed
```

**2. DDR related commands:**

**(1) Query the frequency of DDR:**

```
cat /sys/class/devfreq/dmc/cur_freq
or
cat /sys/kernel/debug/clk/clk_summary | grep ddr
```

**(2) Fix the frequency of DDR (Need extra firmware):**

```
# Query the avilable frequency of DDR
cat /sys/class/devfreq/dmc/available_frequencies
# Fix the frequency of DDR, e.g.,  at 1560MHz
echo userspace > /sys/class/devfreq/dmc/governor
echo 1560000000 > /sys/class/devfreq/dmc/userspace/set_freq
```

**3. NPU related commands:**

**(1) Query the frequency of NPU:**

For RK3566/RK3568:

```
cat /sys/kernel/debug/clk/clk_summary | grep npu
or
cat /sys/class/devfreq/fde40000.npu/cur_freq
```

For RK3588 (**Need to upgrade firmware**):

```
cat /sys/class/devfreq/fdab0000.npu/cur_freq
```

For RV1106/RV1103:

```
cat /sys/kernel/debug/clk/clk_summary | grep npu
```

For RK3562:

```
cat /sys/class/devfreq/ff300000.npu/cur_freq
```

(2) **Fix the frequency of NPU：**

**Note: The driver version after 0.7.2, it is required to turn on the power of NPU first before setting frequency.**

For RK3566/RK3568:

```
# Query the available frequency of NPU
cat /sys/class/devfreq/fde40000.npu/available_frequencies
# Fix NPU frequency， e.g.,  fix at 1 GHz
echo userspace > /sys/class/devfreq/fde40000.npu/governor
echo 1000000000 > /sys/kernel/debug/clk/clk_scmi_npu/clk_rate
```

For RK3588 **(Need to upgrade firmware**) :

```
# Query the available frequency of NPU
cat /sys/class/devfreq/ fdab0000.npu/available_frequencies
# Fix the NPU frequency, e.g., at 1GHz
echo userspace > /sys/class/devfreq/fdab0000.npu/governor
echo 1000000000 > /sys/kernel/debug/clk/clk_npu_dsu0/clk_rate
```

For RV1106/RV1103 **(Unsupported**).

For RK3562:

```
# Query the available frequency of NPU
cat /sys/class/devfreq/ff300000.npu/available_frequencies
# Fix the NPU frequency, e.g., at 600MHz
echo userspace > /sys/class/devfreq/ff300000.npu/governor
echo 600000000 > /sys/class/devfreq/ff300000.npu/userspace/set_freq
```

### 4.8.4 NPU supports query settings

If the NPU driver version is after 0.7.2, you can query the NPU version, the utilization of different NPU cores and manually switch the NPU power supply through the node.

（1）query NPU driver version:

```
cat /sys/kernel/debug/rknpu/driver_version
or
cat /proc/debug/rknpu/driver_version
```

（2）query NPU utilization:

```
cat /sys/kernel/debug/rknpu/load
or
cat /proc/debug/rknpu/load
```

（3）query  NPU power status:

```
cat /sys/kernel/debug/rknpu/power
```

（4）open NPU power:

```
echo on > /sys/kernel/debug/rknpu/power
```

（5）close NPU power:

```
echo off > /sys/kernel/debug/rknpu/power
```

There are some new functions after driver version 0.8.2 shown below.

（1）query NPU frequency:

```
cat /sys/kernel/debug/rknpu/freq
```

（2）Setting up NPU frequency:

```
# using RK3588 as an example
# look up available frequency of NPU
cat /sys/class/devfreq/ fdab0000.npu/available_frequencies
# Fix NPU frequency, for example, 1GHz
echo 1000000000 > /sys/kernel/debug/rknpu/freq
```

（3）query  NPU voltage status:

```
cat /sys/kernel/debug/rknpu/volt
```

（4）query NPU delay time of power during shutdown (unit: ms):

```
cat /sys/kernel/debug/rknpu/delayms
```

（5）Setting up delay time of power during shutdown for NPU Power (unit: ms):

```
echo 2000 > /sys/kernel/debug/rknpu/delayms
```