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

Rockchip User Guide RKNN API

(Technology Department, Graphic Compute Platform Center)

Mark:	Version:	1.7.5
[ ] Changing	Author:	HPC
[√] Released	Completed Date:	31/Jul/2023
	Reviewer:	Vincent
	Reviewed Date:	31/Jul/2023

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

Revision History

Version	Modifier	Date	Modify Description	Reviewer
v0.9.7	Yang Huacong	25/Jan/2019	Initial version	Zhuo Hongtian
V0.9.8	Yang Huacong	5/Jun/2019	1. Add RKNN API Library description 2. Update rknn example and rknn-toolkit path 3. Fix some mistake	Zhuo Hongtian
V1.3.3	Randall	2/June/2020	1. Support RV1109/RV1126 2. Remove the python support instructions, see the RKNN Toolkit Lite related documentation for python support.	Randall
V1.4.0	Randall	10/Sep/2020	1. Add some error code	Randall
v1.6.0	Chifred Hong	15/Jan/2021	1. Add Matmul advanced API 2. Add a set of API for setting inputs and a set of API for obtaining outputs 3. Add quantization and dequantization algorithm description 4. Add NPU driver description 5. Add FAQ	Randall
v1.6.1	Chifred Hong	26/Apr/2021	1. Update Matmul advanced API	Randall
v1.7.0	Chifred Hong	5/Aug/2021	1. Update Version number	Randall
v1.7.1	Chifred Hong	2/Dec/2021	1. Single-channel input performance optimization 2. Fix bug	Randall
v1.7.3	Chifred Hong	13/Aug/2022	1. Update Version number	Randall
V1.7.5	HPC Team	31/Jul/2023	1. Updated the instructions for using the zero-copy interface; 2. Update customer FAQ; 3. Matmul operator supports transposition function, add rknn_matmul_load2 interface.	Randall

Table of Contents

1 Overview

RKNN SDK provides a programming interface for platforms with NPU such as RK1808, which can help users deploy RKNN models exported using RKNN-Toolkit.

2 Supported hardware platforms

This document applies to the following hardware platforms:

1) RK1808, RK1806

2) RV1126, RV1109

The following description takes RK1808 as an example and also applies to other platforms mentioned above.

3 Instructions

3.1 RKNN SDK Development Process

Before using the RKNN SDK, users first need to use the RKNN-Toolkit tool to convert the user's model to the RKNN model. The user can obtain the tool's complete installation package and documentation at https://github.com/rockchip-linux/rknn-toolkit .

After successful conversion to the RKNN model, users can first connect to the RK1808 device via RKNN-Toolkit for online debugging to ensure that the accuracy and performance of the model meet the requirements.

After getting the RKNN model file, users can choose using C or Python interface to develop the application. The following chapters will explain how to develop application based on the RKNN SDK on RK1808 platform.

3.2 RKNN C API

3.2.1 RKNN API Library

The libraries and header files provided by the RKNN SDK are located at <sdk>/rknpu/rknn/rknn_api/librknn_api directory, developers can use to develop applications.

It should be noted that the RKNN API of the RK1808 and RK3399Pro platforms is compatible, and applications developed by both can be easily ported. However, you need to pay attention to distinguish between the two platforms librknn_api.so, if the developer uses RK3399Pro's librknn_api.so will not be able to run on the RK1808 platform. Developers can use the following methods to get the version of librknn_api.so.

```
# librknn_api.so for rk1808 platform
$ strings librknn_api.so |grep version
librknn_api version 1.7.5 (71009c1 build: 2023-07-31 10:30:14)
```

3.2.2 EXAMPLES

The SDK provides MobileNet image classification, MobileNet SSD object detection, and Yolo v3 object detection demos. These demos provide reference for developer to develop applications based on the RKNN SDK. The emo code is located in the <sdk>/external/rknpu/rknn/rknn_api/examples directory. Let's take rknn_mobilenet_demo as an example to explain how to get started quickly.

1) Compile Demo Source Code

```
cd examples/rknn_mobilenet_demo
# modify `GCC_COMPILER` on `build.sh` for target platform, then execute
./build.sh
```

2) Deploy to the RK1808 device

```
adb push install/rknn_mobilenet_demo /userdata/
```

3) Run Demo

```
adb shell
cd /userdata/rknn_mobilenet_demo/
./rknn_mobilenet_demo model/mobilenet_v1_rk180x.rknn
model/dog_224x224.jpg   # RK180x
./rknn_mobilenet_demo model/mobilenet_v1_rv1109_rv1126.rknn
model/dog_224x224.jpg   # RV1109/RV1126
```

3.2.3 API process description

Currently, there are two sets of APIs available on RK1806/RK1808/RV1109/RV1126, which are general API and zero-copy process API. The difference between the two groups of APIs is that each time the general interface updates the frame data, it needs to copy the data allocated by the external module to the input memory of the NPU runtime, while the interface of the zero-copy process will directly use the pre-allocated memory (using the DRM framework).), reducing the cost of memory copies. When the input data has only a virtual address, only the general API interface can be used; when the input data has a physical address or fd, both sets of interfaces can be used.

For the general API interface, first initialize the rknn_input structure, the frame data is included in the structure, call the rknn_inputs_set interface to set the model input, and after the inference is completed, call the rknn_outputs_get function to obtain the inference output result for post-processing. Before each inference, update the frame data. The general API calling process is shown in the following figure:

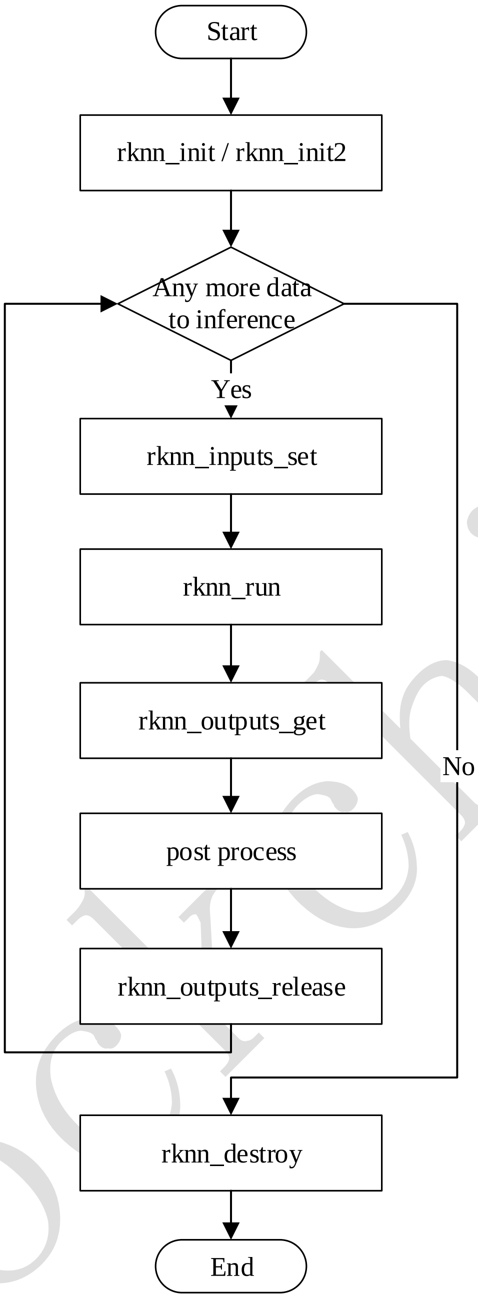

Figure 3-1 General API calling process

For the zero-copy interface, when setting the memory used for input and output, call the rknn_create_mem interface to create memory. The interface will allocate a piece of physical memory through the DRM framework according to the set memory size, and save the physical address, virtual address and file descriptor (fd) corresponding to the physical memory in the rknn_tensor_mem structure. After assigning the address, establish the relationship between the input and output memory and the model input and output nodes through the rknn_set_io_mem interface. After completing these two steps, the application program can update the input data of the model by writing data to the input memory. After the model inference is completed, the result will be automatically updated to the physical memory of the output node. Take the data and complete the corresponding processing. After the application ends, the rknn_destroy_mem interface needs to be called to release the previously allocated memory. In this way, in certain scenarios, the number of memory copies in the model inference process can be reduced, and the overall operating efficiency of the model can be improved. A typical zero-copy interface usage flow is shown in the following figure:

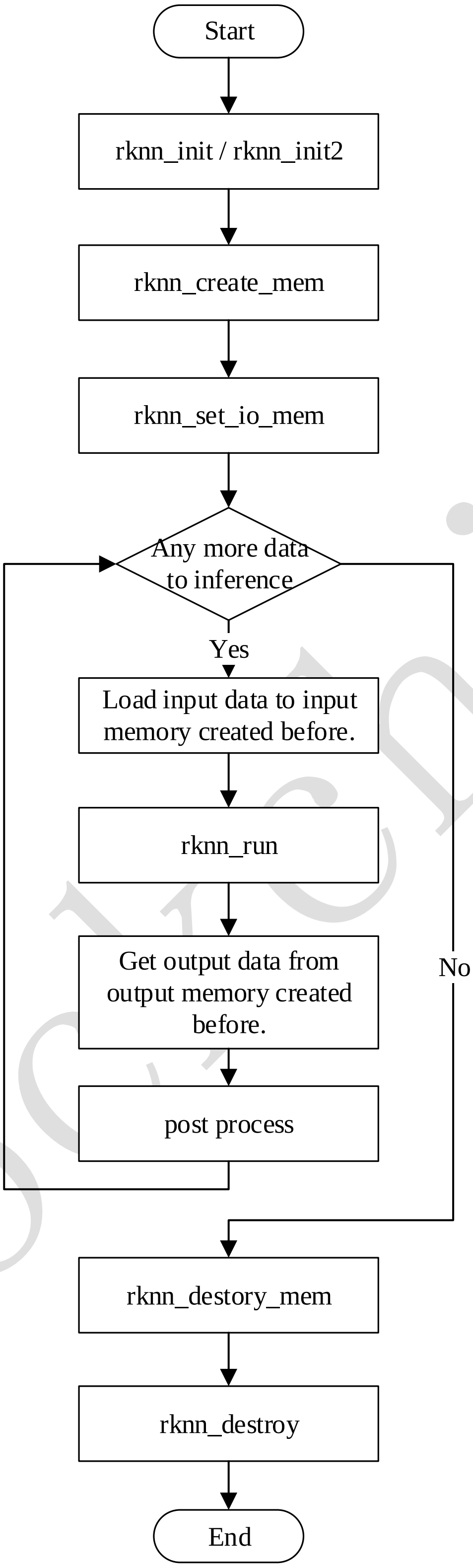

Figure 3-2 Zero-copy interface usage process

The input and output of the model are separate parts, so you can mix the normal set/get interface and the rknn_set_io_mem interface. For example, for the output of the model, use the rknn_outputs_get interface, but for the input of the model, use the rknn_create_mem interface to create the input memory, and use the rknn_set_io_mem interface to establish an association with the model input. The typical interface mixing process is as follows：

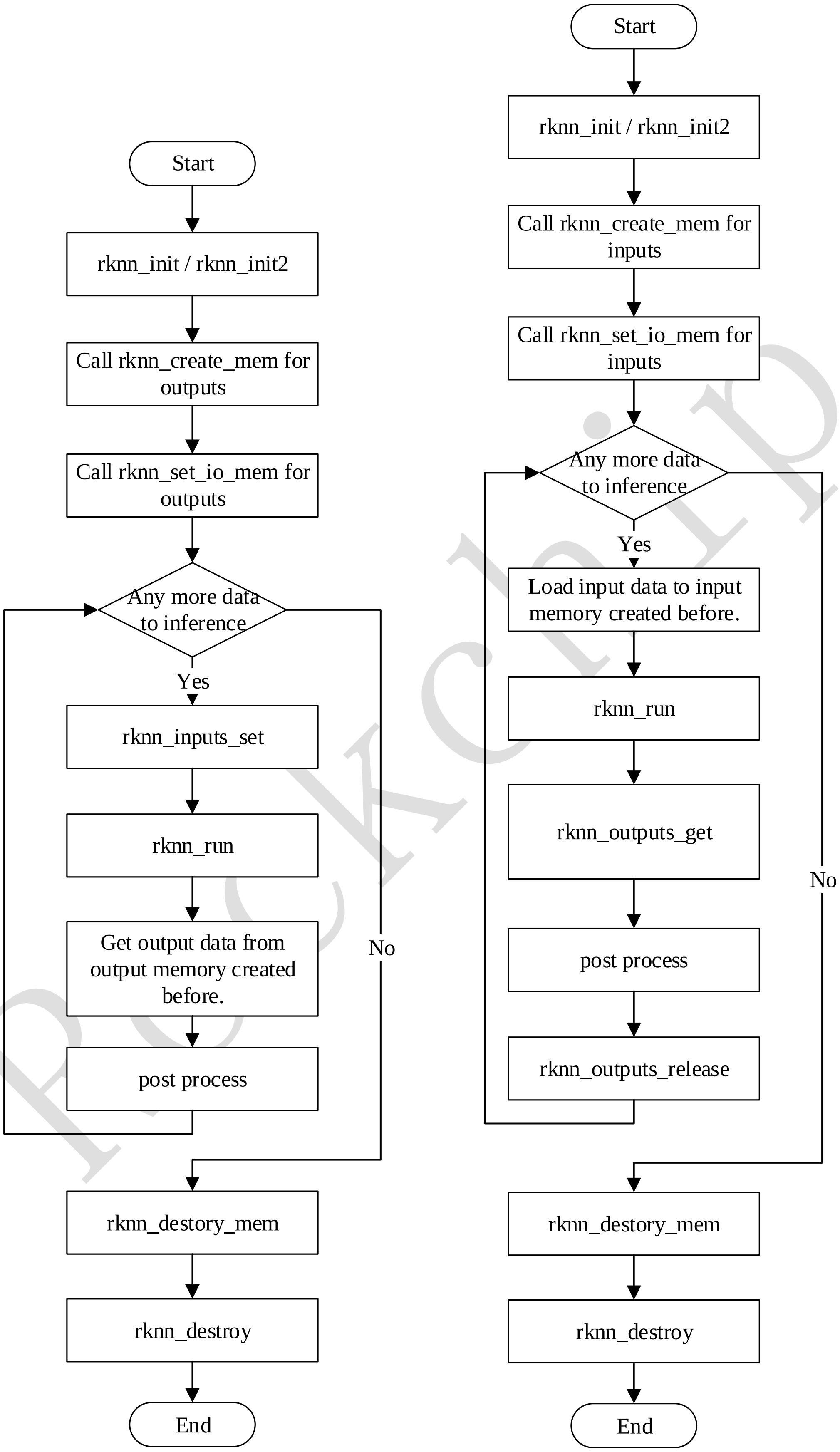

Figure 3-3 A mix of generic API interfaces and zero-copy interfaces

3.2.3.1 *API* internal *processing flow*

During inference of RKNN model, the original data has to go through three processes: input processing, NPU running model, and output processing. In a typical picture inference scenario, assuming that the input data is a 3-channel picture and is in the NHWC layout format, the data processing flow at runtime is shown in Figure 3-4. At the API level, the rknn_inputs_set interface (when pass_through=0, see the rknn_input structure for details) includes the process of swapping color channel, normalization, quantization, and conversion of NHWC to NCHW. The rknn_outputs_get interface (when want_float=1, see rknn_output structure) contains the process of dequantization.

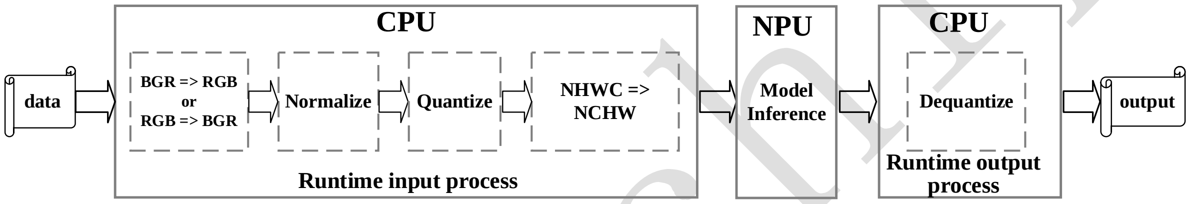

Figure 3-4 Normal mode image data processing flow

In fact, for some RKNN models, the Runtime input processing flow is not all executed. For example, when the input data is not a 3-channel image or the config function of the rknn-toolkit is configured as reorder="0 1 2" when exporting model, there is no color channel swapping process. When the attribute of RKNN model input tensor is NHWC layout, there is no process for converting NHWC to NCHW. rknn_inputs_set (when pass_through=1) and *rknn_inputs_map* do not contain any input processing flow, rknn_outputs_get (when want_float=0) and *rknn_outputs_map* do not contain any output processing flow. At this time, although the two sets of APIs do not include the corresponding processing flow, using the set/get series interface will have more data copying process than the map series interface.

3.2.3.2 Quantification and dequantization

When using rknn_inputs_set (pass_through=1) and *rknn_inputs_map*, it indicates that the process before NPU inference needs to be processed by the user. After *rknn_outputs_map* gets the output, the user also needs to dequantize output to get the 32-bit floating point result.

The quantization method, quantization data type and quantization parameters used in quantization and dequantization can be queried through the rknn_query interface. Currently, the NPU of RK1808/RK3399Pro/RV1109/RV1126 has two quantization methods: asymmetric quantization and dynamic fixed-point quantization. Each quantization method specifies the corresponding quantized data type. There are a total of four combinations of data types and quantification methods:

- uint8 (asymmetric quantization)
- int8 (dynamic fixed-point)
- int16 (dynamic fixed-point)
- float16 (none)

Normally, 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, please refer to the IEEE-754 standard. Assuming that the normalized 32-bit floating point data is D, the following describes the quantization process:

1) float32 to uint8

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

$$D_q = round(clamp(D / S_q + ZP, 0, 255)) \quad (3\text{-}1)$$

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

2) float32 to int8

Assuming that the dynamic fixed-point quantization parameter of the input tensor is fl, the data quantization process is expressed as the following formula:

$$D_q = round(clamp(D * 2^{fl}, -128, 127)) \qquad (3\text{-}2)$$

3) float32 to int16

Assuming that the dynamic fixed-point quantization parameter of the input tensor is fl, the data quantization process is expressed as the following formula:

$$D_q = round(clamp(D * 2^{fl}, -32768, 32767)) \qquad (3\text{-}3)$$

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

3.2.3.3 Zero copy

In specific case, the number of input data copies can be reduced to 0, that is, zero copy. For example, when the RKNN model is asymmetric quantization, the quantization data type is uint8, the mean value of the 3 channels is the same integer and the scaling factor is the same, the normalization and quantization can be omitted. The proof is as follows:

Assume that the input image data is D_f, and the quantization parameter is S_q, ZP. M_i Represents the mean value of the i-th channel, and S_i represents the normalization factor of the i-th channel. Then the normalized data of the i-th channel is as follows:

$$D_i = (D_f - M_i) / S_i \qquad (3\text{-}4)$$

The data quantification process is expressed as the following formula:

$$D_q = clamp(D_i / S_q + ZP, 0, 255) \qquad (3\text{-}5)$$

After combining the above two formulas, we can get

$$D_q = clamp((D_f - M_i)/(S_i * Sq) + ZP) \qquad (3\text{-}6)$$

Assuming that the data range of the calibration data set contains integer values from 0 to 255, when $M_1 = M_2 = M_3$, $S_1 = S_2 = S_3$, the normalized value range is expressed as follows:

$$D_{min} = (0 - M_i) / S_i = -M_i / S_i \qquad (3\text{-}7)$$

$$D_{max} = (255 - M_i) / S_i \qquad (3\text{-}8)$$

So, the quantization parameter is calculated as follows:

$$S_q = (D_{\max} - D_{\min}) / 255 = 1/S_i \qquad (3\text{-}9)$$

$$ZP = (0 - D_{\min}) / S_q = M_i \qquad (3\text{-}10)$$

Substituting formula (3-9) and formula (3-10) into formula (3-6), it can be concluded that under the condition of zero-copy: the mean value of the 3 channels is the same integer and the normalized scaling factor is the same , The input uint8 data is equal to the quantized uint8 data.

Input zero copy can reduce the CPU load and improve the speed of complete inference. For RGB or BGR input data, the steps to achieve input zero-copy are as follows:

1) The mean value of the three channels is the same integer and the normalized scaling factor is the same.

2) In the *config* function of rknn-toolkit, set *force_builtin_perm=True* to export the RKNN model input by NHWC.

3) Call the rknn_create_mem interface to apply for physical memory for the input tensor and save the memory address information.

4) When using it for the first time, call the rknn_set_io_mem interface to create an association between the input memory and the output tensor.

5) Fill the memory address with input data, such as calling the RGA resize function, the target address uses the physical address obtained by *rknn_inputs_map*

6) Call the rknn_run interface.

7) Call the get output interface.

3.2.4 API Reference

3.2.4.1 rknn_init

The *rknn_init* function will create a *rknn_context* object, load the RKNN model, and perform specific initialization behavior based on the flag.

API	rknn_init
Description	Initialize rknn
Parameters	*rknn_context *context*：Pointer to *rknn_context* object. After the function is called, the context object will be assigned.
	*void *model*：Binary data for the RKNN model.
	uint32_t size：Model size
	uint32_t flag：A specific initialization flag. Currently supports following flags: **RKNN_FLAG_COLLECT_PERF_MASK**：Open the performance collection debugging switch. After opening, you can query the running time of each layer of the network through the *rknn_query* interface. Note that the running time of *rknn_run* will be longer after this flag is set.
Return	int: Error code (See RKNN Error Code).

Sample Code:

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

3.2.4.2 rknn_destroy

The *rknn_destroy* function will release the *rknn_context* object and its associated resources.

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

Sample Code:

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

3.2.4.3 rknn_query

The *rknn_query* function can query the information of model input and output tensor attribute, performance information and SDK version etc.

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 cmd: Query command.
	void info*: 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 the 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 Tensor.
RKNN_QUERY_INPUT_ATTR	rknn_tensor_attr	Query input Tensor attribute.
RKNN_QUERY_OUTPUT_ATTR	rknn_tensor_attr	Query output Tensor attribute.
RKNN_QUERY_PERF_DETAIL	rknn_perf_detail	Query the running time of each layer of the network.
RKNN_QUERY_SDK_VERSION	rknn_sdk_version	Query the SDK version.

Next we will explain each query command in detail.

1） 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);
```

2） Query input Tensor attribute

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.

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));
}
```

3） Query output Tensor attribute

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));
}
```

4） Query the running time of each layer of the network

If the *RKNN_FLAG_COLLECT_PERF_MASK* flag has been set on the *rknn_init* function, the *RKNN_QUERY_PERF_DETAIL* command can be passed to query the runtime of each layer of the network after *rknn_run* function execution completed. You need to create the *rknn_perf_detail* structure object first.

Sample Code:

```
rknn_perf_detail perf_detail;
ret = rknn_query(ctx, RKNN_QUERY_PERF_DETAIL, &perf_detail,
                              sizeof(rknn_perf_detail));
printf("%s", perf_detail.perf_data);
```

It should be noted that *rknn_perf_detail.perf_data* does not need to be released. The SDK will automatically manage this buffer memory.

5） 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);
```

3.2.4.4 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 one is a *rknn_input* structure object. Developers needs to set these object field before passing in.

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;

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

3.2.4.5 rknn_inputs_map(deprecated)

This interface is deprecated. Please use the rknn_create_mem and rknn_set_io_mem interfaces to create input memory and establish an association with the model input node. If you want to continue to use this interface, please refer to the user guide document of version 1.7.3.

3.2.4.6 rknn_inputs_sync(deprecated)

This interface is deprecated. Please use the rknn_create_mem and rknn_set_io_mem interfaces to create input memory and establish an association with the model input node. If you want to continue to use this interface, please refer to the user guide document of version 1.7.3.

3.2.4.7 rknn_inputs_unmap(deprecated)

This interface is deprecated. Please use the rknn_destroy_mem interface to delete the memory created for the input node. If you want to continue to use this interface, please refer to the user guide document of version 1.7.3.

3.2.4.8 rknn_run

The *rknn_run* function will perform a model reasoning. The input data need to be set by the *rknn_inputs_set* function before *rknn_run is* called.

API	rknn_run
Description	Perform a model reasoning.
Parameter	*rknn_context context*: The object of rknn_contex.
	rknn_run_extend extend*: Reserved for extension, currently not used, you can pass NULL.
Return	int: Error code (See RKNN Error Code).

Sample Code:

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

3.2.4.9 rknn_outputs_get

The *rknn_outputs_get* function can get the output data of the model reasoning. This function can get multiple output data. Each of these outputs is a *rknn_output* structure object, which needs to be created and set in turn before the function is called.

There are two ways to store buffers for output data:

1) Developer allocate and release buffers themselves. At this time, 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 SDK. 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, you can pass NULL.
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].want_float = 1;
}
ret = rknn_outputs_get(ctx, io_num.n_output, outputs, NULL);
```

3.2.4.10 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[]: rknn_output array to be release.
Return	int: Error code (See RKNN Error Code).

Sample Code:

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

3.2.4.11 rknn_outputs_map(deprecated)

This interface is deprecated. Please use the rknn_create_mem and rknn_set_io_mem interfaces to create output memory and establish an association with the model output node. If you want to continue to use this interface, please refer to the user guide document of version 1.7.3.

3.2.4.12 rknn_outputs_sync(deprecated)

This interface is deprecated. Please use the rknn_create_mem and rknn_set_io_mem interfaces to create output memory and establish an association with the model output node. If you want to continue to use this interface, please refer to the user guide document of version 1.7.3.

3.2.4.13 rknn_outputs_unmap(deprecated)

This interface is deprecated, please use the rknn_destroy_mem interface to delete the memory created for the output node. If you want to continue to use this interface, please refer to the user guide document of version 1.7.3.

3.2.4.14 rknn_create_mem

This interface can allocate a piece of physical memory for saving the data input or output of the model.

API	rknn_create_mem
Description	Create a rknn_tensor_mem structure and allocate memory of the specified size.
Parameter	rknn_context context: rknn_context object.
	uint32_t size: The size of the memory to be allocated.
Return	rknn_tensor_mem*: pointer for tensor memory information structure.

Sample Code:

```
    rknn_tensor_mem* input_mems[1];
    input_mems[0] = rknn_create_mem(ctx, size);
```

3.2.4.15 rknn_set_io_mem

This interface can associate the memory allocated by rknn_create_mem with the input and output nodes of the model, so that the NPU can use this memory to obtain input or save output data.

API	rknn_set_io_mem	
Description	Sets the rknn_tensor_mem structure containing the model input (or output) memory information.	
Parameter	rknn_context context: rknn_context object.	
	rknn_tensor_mem*: pointer for tensor memory information structure.	
	rknn_tensor_attr *: input (or output) attributes of the tensor.	
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_OUTPUT_ATTR, &(output_attrs[0]),
sizeof(rknn_tensor_attr));

    output_mems[0] = rknn_create_mem(ctx, output_attrs[0].size);

    rknn_set_io_mem(ctx, output_mems[0], &output_attrs[0]);
```

3.2.4.16 rknn_destroy_mem

This interface will destroy the memory allocated in the specified rknn_tensor_mem. Usually called before the end of the application to release memory resources and avoid memory leaks.

API	rknn_destroy_mem
Description	Destroy the rknn_tensor_mem structure to release memory resources.
Parameter	rknn_context context: rknn_context object.
	rknn_tensor_mem*: pointer for tensor memory information structure.
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]);
```

3.2.5 RKNN DataStruct Define

3.2.5.1 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

3.2.5.2 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: **RKNN_TENSOR_NCHW** **RKNN_TENSOR_NHWC**
type	rknn_tensor_type	Tensor data type, has the following data types: **RKNN_TENSOR_FLOAT32** **RKNN_TENSOR_FLOAT16** **RKNN_TENSOR_INT8** **RKNN_TENSOR_UINT8** **RKNN_TENSOR_INT16**
qnt_type	rknn_tensor_qnt_type	Tensor Quantization Type, has the following types of quantization: **RKNN_TENSOR_QNT_NONE:** Not quantified; **RKNN_TENSOR_QNT_DFP**: Dynamic fixed point quantization; **RKNN_TENSOR_QNT_AFFINE_ASYMMETRIC**: Asymmetric quantification.
fl	int8_t	RKNN_TENSOR_QNT_DFP quantization parameter

zp	uint32_t	RKNN_TENSOR_QNT_AFFINE_ASYMMETRIC quantization parameter.
scale	float	RKNN_TENSOR_QNT_AFFINE_ASYMMETRIC quantization parameter.

3.2.5.3 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*	Point to 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.

3.2.5.4 rknn_tensor_mem

The structure rknn_tensor_mem represents the storage state information after tensor initialization, which is used as a parameter to pass in to the rknn_inputs_map series and rknn_outputs_map series functions. The definition of the structure is shown in the following table:

Field	Type	Meaning
logical_addr	void*	The virtual address of this input.
physical_addr	uint64_t	The physical address of this input.
fd	int32_t	The fd of this input.
size	uint32_t	The memory size of the input tensor.
handle	uint32_t	The handle of this input.
priv_data	void*	Reserved data.
reserved_flag	int32_t	Reserved flag.

3.2.5.5 rknn_output

The structure *rknn_output* represents a data output of the model, used as a parameter to the *rknn_outputs_get* function. The following table shows the definition:

Field	Type	Meaning
want_float	uint8_t	Indicates if 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*	Pointer which point to the output data Buffer.
size	uint32_t	Output data Buffer memory size.

3.2.5.6 rknn_perf_detail

The structure *rknn_perf_detail* represents the performance details of the model. The following table shows the definition:

Field	Type	Meaning
perf_data	char*	Performance details include the run time of each layer of the network.
data_len	uint64_t	The Length of perf_data.

3.2.5.7 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[]	Driver version information.

3.2.6 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_PRE_COMPILE_MODEL (-11)	This RKNN model use pre_compile mode, but not compatible with current driver.
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 don't compatible with current platform.
RKNN_ERR_NON_PRE_COMPILED_MODEL_ON_MINI_DRIVER(-14)	The RKNN model is not in pre_compile mode and cannot be executed on mini-driver.

4 Advanced API instructions

4.1 Matmul Operator library

4.1.1 Introduction

The high-level API is designed to use the high computing power of the NPU to perform specific mathematical operations, provide a simple interface call, and achieve the effect of computing acceleration. Among them, the Matmul operator library is an acceleration library for fixed-point matrix multiplication. The operation is defined as follows:

$$C = A^T * B$$

Here:

A, B and C are 2-dimensional matrices

A is a K*M matrix

B is a K*N matrix

C is a M*N matrix

4.1.2 Data structure definition

rknn_matmul_handle_t represents the handle used to perform the operation of the Matmul operator, which contains the context of the runtime environment and the information of the input buffer. The definition of the structure is shown in the following table:

Field	Type	Meaning
A	void*	The pointer of the first matrix buffer during operation.
B	void*	The pointer of the second matrix buffer during operation.
M	int32_t	The low rank dimension of A matrix.
K	int32_t	The high rank dimension of A and B matrix.
N	int32_t	The low rank dimension of B matrix.
in_dtype	rknn_tensor_type	The type of input data.
rknn_ctx	rknn_context	The context object at runtime.
transposeA	int8_t	The flag of A matrix transposition, 0 indicates that the A matrix is not transposed, and 1 indicates that the A matrix is transposed before matrix multiplication.
transposeB	int8_t	The flag of B matrix transposition, 0 indicates that the B matrix is not transposed, 1 indicates that the B matrix is transposed before matrix multiplication.

4.1.3 Detailed API description

4.1.3.1 rknn_matmul_load

The rknn_matmul_load loading function will load the input buffer created by the user and return an object of type rknn_matmul_handle_t.The Matmul operator API is not responsible for managing the life cycle of the input buffer, and the user must ensure that the input buffer is valid within the Matmul operator API call.

API	rknn_matmul_load
Description	Initialize and set the input buffer pointer.
Parameter	void *a：The pointer of the first matrix buffer created by the user can only support the input of 8-bit unsigned integer or 8-bit signed integer one-dimensional array pointer.
	void *b：The pointer of the second matrix buffer created by the user only supports the input of 8-bit unsigned integer or 8-bit signed integer one-dimensional array pointer.
	int32_t M: The low rank dimension of A matrix.
	int32_t K: The high rank dimension of A and B matrix.
	int32_t N: The low rank dimension of B matrix.
	rknn_tensor_type dtype：The input data type specified by the user, only supports RKNN_TENSOR_INT8 or RKNN_TENSOR_UINT8 type
Return	rknn_matmul_handle_t object。

The sample code is as follows:

```
rknn_tensor_type dtype = RKNN_TENSOR_INT8;
int8_t x[256*1] = {0};
int8_t y[256*4096] = {0};
rknn_matmul_handle_t handle= rknn_matmul_load(x,y,1,256,4096,dtype);
```

4.1.3.2 rknn_matmul_load2

The rknn_matmul_load2 function adds the feature of matrix multiplication with transposing B matrix on the basis of the rknn_matmul_load function.

API	rknn_matmul_load2
Description	Initialize and set the input buffer pointer.
Parameter	void *a：The pointer of the first matrix buffer created by the user can only support the input of 8-bit unsigned integer or 8-bit signed integer one-dimensional array pointer.
	void *b：The pointer of the second matrix buffer created by the user only supports the input of 8-bit unsigned integer or 8-bit signed integer one-dimensional array pointer.
	int32_t M: The low rank dimension of A matrix.
	int32_t K: The high rank dimension of A and B matrix.
	int32_t N: The low rank dimension of B matrix.
	rknn_tensor_type dtype：The input data type specified by the user, only supports RKNN_TENSOR_INT8 or RKNN_TENSOR_UINT8 type
	transposeA: The flag of A matrix transposition, 0 indicates that the data arrangement of A matrix is [K, M], currently only supports transposeA=0, M=1.
	transposeB: The flag of B matrix transposition, 0 indicates that the data arrangement of B matrix is [K,N], 1 indicates that the B matrix data arrangement is [N,K].
Return	rknn_matmul_handle_t object。

The sample code is as follows:

```
rknn_tensor_type dtype = RKNN_TENSOR_INT8;
int8_t x[256*1] = {0};
int8_t y[256*4096] = {0};
rknn_matmul_handle_t handle=
        rknn_matmul_load2(x,y,1,256,4096,dtype,0,1);
```

4.1.3.3 rknn_matmul_run

After rknn_matmul_load/rknn_matmul_load2 is called and before rknn_matmul_run is executed, the data in the input buffer is updated externally, without calling rknn_matmul_load/rknn_matmul_load2 again.

API	rknn_matmul_run
Description	Perform Matmul operations.
Parameter	rknn_matmul_handle_t matmul_handle: The handle returned by the rknn_matmul_load interface.
	float *c: The pointer to the matrix floating-point buffer created by the user, used to obtain the output.
Return	int: Error code (See RKNN Error Code).

The sample code is as follows:

```
...
float out_fp32_buf[4096] = {0};
rknn_matmul_run(handle,out_fp32_buf);
```

4.1.3.4 rknn_matmul_unload

Both the contexts initialized by the rknn_matmul_load or rknn_matmul_load2 interfaces is destroyed using rknn_matmul_unload.

API	rknn_matmul_unload
Description	Destroy the Matmul operator runtime context.
Parameter	rknn_matmul_handle_t matmul_handle: The handle returned by the rknn_matmul_load interface.
Return	int: Error code (See RKNN Error Code).

The sample code is as follows:

```
...
rknn_matmul_unload(handle);
```

4.1.4 Implementation restrictions

The Matmul operator library is implemented based on the hardware architecture of the NPU. In order to achieve a balance between accuracy and speed, there are some restrictions as follows.

4.1.4.1 Dimensional restrictions

According to the above operation description, the library realizes the matrix multiplication of M=1. Specifically, the input A of the Matmul operator must be a Kx1 buffer(row-major), that is, the user must create a piece of data that contains K 8-bit unsigned integers or 8-bit signed integers. When the operator library is running on the Mini driver, the value of K can only be set to 128 or 256 or 512, and N is fixed at 4096. When running on the Full driver, there is no such limit, but the recommended number of K is 128, 256, 512, 1024, 2048, N is an even power of 2. It is recommended that N should not be greater than 4096.

4.1.4.2 Other restrictions

Currently, the rknn_matmul_load2 interface is only supported on the Full driver, and transposeA must be 0.

4.1.4.3 Input data type restriction

Only supports 8-bit unsigned integer and 8-bit signed integer input.

4.1.5 Benchmark

When the two input matrices use random numbers, the measured results on the RV1109-EVB board are shown in Table-1. The speed is the average time after the rknn_matmul_run interface is called 100 times. The average relative error is the error value of the result of executing the same algorithm on the NPU and CPU. The specific formula is:

$$\sum_k (abs(R_1 - R_2)/R_2)/N$$

among them,

R_1 is the output vector of the Matmul operator library, containing N elements.

R_2 is the CPU output vector, containing N elements.

Table-1 Speed/accuracy results of Matmul operator library (RV1109, int8, transposeA=0, transposeB=0)

K	N	Speed（ms）	Average relative error
128	1024	1.0	0.00034
256	1024	1.6	0.00032
512	2024	3.0	-0.00015
1024	1024	5.4	0.00047
128	4096	3.0	0.00051
256	4096	5.6	0.00024
512	4096	10.7	0.00024
1024	4096	20.9	0.00051

Note: the speed may vary slightly due to different NPU driver versions. The error value may varies slightly according to the random number of each test.

5 NPU driver description

5.1.1 Directory structure description

The NPU driver is in the **$SDK/external/rknpu/drivers/** directory or

https://github.com/rockchip-linux/rknpu/tree/master/drivers

The compilation and installation rules refer to：

$SDK/buildroot/package/rockchip/rknpu/rknpu.mk

```
drivers/
├── common
├── linux-aarch64 （for RK1808 npu full driver）
├── linux-aarch64-mini （for RK1808 npu mini driver）
├── linux-armhf （for RK1806 npu full driver）
├── linux-armhf-mini（for RK1806 npu mini driver）
├── linux-armhf-puma （for RV1126/RV1109 npu full driver）
├── linux-armhf-puma-mini（for RV1126/RV1109 npu mini driver）
├── npu_ko （NPU kernel driver）
```

5.1.2 The difference between NPU full driver and mini driver

Include the following points:

1) Mini driver only supports pre-compiled rknn model. If you run non-pre-compiled model, RKNN_ERR_MODEL_INVALID error will appear. Starting from 1.6.0, it will return RKNN_ERR_NON_PRE_COMPILED_MODEL_ON_MINI_DRIVER error;

2) The full driver supports the online debugging function of RKNN Toolkit, but the mini driver does not;

3) Mini driver library size is much smaller than full driver. Taking RV1109/RV1126 1.6.0 driver as an example, full driver size is 87MB, mini driver size is 7.1MB, which can effectively save flash size.

4) Mini driver library occupies less memory than full driver when running.

6 FAQ

6.1.1 Input and output data format issues

6.1.1.1 How to choose from RGB or BGR format when given three-channel image data input?

It is recommended that using RGB format input data uniformly. When exporting the RKNN model, there are two possibilities for the reorder_channel parameter of the config function:

1) If the original model is trained using BGR images, reorder_channel = '2 1 0'.

2) If the original model is trained using RGB images, reorder_channel = '0 1 2'.

6.1.1.2 Which RKNN_TENSOR_NHWC or RKNN_TENSOR_NCHW should be set in the rknn_input structure? Why are the two settings time-consuming different?

The rknn_input structure is determined according to the user's own data format, and the C API will automatically convert it into the format required by the NPU.

The reason for the different time-consuming is that different input formats have different calculation amounts and different optimization methods.

6.1.1.3 For the non-quantized RKNN model, why is the size in the output rknn_tensor_attr different from the size of the rknn_output returned by the rknn_outputs_get interface?

The non-quantized RKNN model, the internal output data type of the NPU is float16, and the size is the number of elements * 2 bytes. When the user sets want_float = 1, what they want is float32 data, float16 will be converted to float32, and the size is the number of elements * 4 bytes.

6.1.1.4 Is rknn_output.index set by user or return by driver?

Return by driver.

6.1.1.5 Why does the dims array of the rknn_tensor_attr have 0?

0 means the size is invalid. n_dims in rknn_tensor_attr represents the number of valid size in dims.

6.1.1.6 The order of dims in rknn_tensor_attr is opposite to the order of numpy obtained by rknn_toolkit?

The layout of arrays in C API is the opposite of python. For example, the numpy output shape obtained by the run() interface of rknn-toolkit is [1,255,20,20], and the dims array in C API is {20,20,255,1}.

6.1.2 Input and output interface usage problems

6.1.2.1 How to preprocess the data when using pass_through and using rknn_inputs_map interface?

Please refer to the rknn_pass_through_demo example under https://github.com/rockchip-linux/rknpu/tree/master/rknn/rknn_api/examples.

6.1.2.2 Why is the physical address obtained using rknn_inputs_map or rknn_outputs_map invalid? How to obtain a valid physical address?

Input/output cannot be allocated to physically contiguous memory. The possible reasons are:

1) The input/output size is too large, exceeding the total physically contiguous memory size (the default is 4MB).

2) There is not enough physically contiguous memory available in the system.

3) When exporting the RKNN model, add the following parameter to the config function:

output_optimize=1.

Users can try to restart the system, or configure a larger physically continuous memory space when the NPU driver is mounted. For the configuration method, refer to the rknn_inputs_map interface description.

Note: This group of interfaces has been abandoned. Please use the newly added rknn_create_mem to create memory for input and output, and use the rknn_set_io_mem interface to establish the relationship between input and output memory and the RKNN model.

6.1.3 API call process issues

6.1.3.1 After rknn_init succeeds, can the memory occupied by the model file be released?

Yes,you can.

6.1.3.2 When rknn_output.is_prealloc=1, does rknn_outputs_release need to be called?

Yes, it needs.

6.1.3.3 In a project which need run 2 models at the same time, so it defines two rknn_contexts to obtain output1 and output2 respectively. Does it only need to perform rknn_outputs_release once when releasing?

To call the rknn_outputs_release interface twice, release the memory of output1 and output2 respectively. Be careful to use a different context when calling.

6.1.4 Performance issues

6.1.4.1 rknn_init takes too long?

Use pre-compiled models. Refer to the relevant chapters of the User Guide document under https://github.com/rockchip-linux/rknn-toolkit/tree/master/doc for usage.

6.1.4.2 rknn_inputs_set takes too long?

The possible reason is the large amount of data or the time-consuming format conversion. If it takes a long time to convert the format, users can try the pass_through usage to do the conversion themselves. For conversion method, please refer to rknn_pass_through_demo example under https://github.com/rockchip-linux/rknpu/tree/master/rknn/rknn_api/examples, or try to export the RKNN model by adding the following parameter to the config function: output_optimize=1.

6.1.4.3 rknn_outputs_get takes too long?

The possible reason is the large amount of data or the time-consuming format conversion. If it takes a long time to convert the format, users can try to set want_float=0 and do the conversion themselves, or try to export the RKNN model by adding the following parameter to the config function: output_optimize=1.

6.1.4.4 Is there a big difference in speed between Python API and C API calls?

The Python API is essentially the encapsulation of the C API, with additional overhead and slower speed. Moreover, it does not support features such as zero copy, which may be limited in performance optimization. If the performance requirements are relatively high, it is not recommended to use the Python API.

6.1.5 Demo issues

6.1.5.1 Why do some examples only support images in bmp format? How can I support images in other formats?

In examples such as rknn_yolov5_demo, in order to draw the detection result in the image, the method in the CImg library is used, and this library only supports pictures in bmp format.

If you want to support images in other formats, you can use graphics libraries such as OpenCV, and the code needs to be changed accordingly. If you don't need to draw the result image, you can also remove the CImg-related code, and the stb_image library used to read the image supports other image formats such as JPEG/PNG.

6.1.5.2 How to get the cross-compilation toolchain

For the RV1109 / RV1126 platform, please download the cross compilation tool through the link below:

https://developer.arm.com/-/media/Files/downloads/gnu-a/8.3-2019.03/binrel/gcc-arm-8.3-2019.03-x86_64-arm-linux-gnueabihf.tar.xz?revision=e09a1c45-0ed3-4a8e-b06b-db3978fd8d56&rev=e09a1c450ed34a8eb06bdb3978fd8d56&hash=9C4F2E8255CB4D87EABF5769A2E65733

For the RK1808 platform, please download the cross-compilation tool through the link below:

https://releases.linaro.org/components/toolchain/binaries/6.3-2017.05/aarch64-linux-gnu/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu.tar.xz

For the RK1806 platform, please download the cross-compilation tool through the link below:

https://releases.linaro.org/components/toolchain/binaries/6.3-2017.05/arm-linux-gnueabihf/gcc-linaro-6.3.1-2017.05-x86_64_arm-linux-gnueabihf.tar.xz

6.1.6 Zero-copy issues

6.1.6.1 Using the zero-copy interface, will it affect the accuracy?

In theory not. Accuracy problems occur, usually caused by incorrect processing of the input data.

6.1.6.2 How to fix memory allocation fails when using the zero-copy interface?

If the memory required for model input and output is relatively large, exceeding 4MB, the problem that physical_addr is an illegal address may occur when calling the old zero-copy interface rknn_inputs_map.

This happens because the old zero-copy interface uses DMA memory reserved by the system, which is very small.

In this case, it is recommended to use the new zero-copy interface. After version 1.7.3, all zero-copy examples will use the new zero-copy interface.

6.1.7 Driver issues

6.1.7.1 What is the difference between galcore_puma.ko and galcore_puma_tb.ko

The ko file is associated with the Linux kernel of the system, and the kernel of different development boards will be different. galcore_puma_tb.ko is the driver that matches the kernel of Rockchip RV1109/RV1126 Toybrick board (tb is the abbreviation of Toybrick), and galcore_puma.ko matches the driver of the kernel of Rockchip RV1109/RV1126 EVB development board. If it is a board from another manufacturer, please contact the corresponding manufacturer to obtain ko.