Arm China Zhouyi Compass Driver and Runtime

Version: 1.0

User Guide

Confidential

Arm China Zhouyi Compass Driver and Runtime

User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0001-00	30 September 2023	Confidential	First release for 1.0

Confidential Proprietary Notice

This document is **CONFIDENTIAL** and any use by you is subject to the terms of the agreement between you and Arm Technology (China) Co., Ltd ("Arm China") or the terms of the agreement between you and the party authorized by Arm China to disclose this document to you.

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm China. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information: **(i)** for the purposes of determining whether implementations infringe any third party patents; **(ii)** for developing technology or products which avoid any of Arm China's intellectual property; or **(iii)** as a reference for modifying existing patents or patent applications or creating any continuation, continuation in part, or extension of existing patents or patent applications; or **(iv)** for generating data for publication or disclosure to third parties, which compares the performance or functionality of the Arm China technology described in this document with any other products created by you or a third party, without obtaining Arm China's prior written consent.

THIS DOCUMENT IS PROVIDED "AS IS". ARM CHINA PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm China makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM CHINA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM CHINA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm China's customers is not intended to create or refer to any partnership relationship with any other company. Arm China may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm China, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm China is a trading name of Arm Technology (China) Co., Ltd. The words marked with ® or ™ are registered trademarks in the People's Republic of China and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners.

Confidentiality Status

This document is Confidential. This document may only be used and distributed in accordance with the terms of the agreement entered into by Arm China and the party that Arm China delivered this document to.

Product Status

The information in this document is Final, that is a for a developed product.

Web Address

https://www.armchina.com

Contents
Arm China Zhouyi Compass Driver and Runtime User Guide

Preface

This preface introduces the *Arm China Zhouyi Compass Driver and Runtime User Guide*.

It contains the following sections:

- *About this book* on page 6.
- *Feedback* on page 8.

About this book

This book describes how the Zhouyi Compass Diver and Runtime work and how to use them.

Using this book

This book is organized into the following chapters:

Chapter 1 NPU driver

This chapter describes the driver of Zhouyi NPUs.

Chapter 2 NPU runtime

This chapter describes the runtime of Zhouyi NPUs.

Glossary

The Arm® Glossary is a list of terms used in Arm China documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm China meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic

Introduces special terminology, denotes cross-references, and citations.

bold

Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`

Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace` (with "mono" underlined)

Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`

Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

`<and>`

Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Additional reading

This book contains information that is specific to this product. See the following documents for other relevant information.

Arm China publications

The following confidential documents are only available to licensees:

- *Arm China Zhouyi Compass Getting Started Guide*.
- *Arm China Zhouyi Compass Software Technical Overview*.
- *Arm China Zhouyi Compass NN Compiler User Guide*.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content, send an e-mail to *errata@armchina.com*. Give:

- The title *Arm China Zhouyi Compass Driver and Runtime User Guide*.
- The number 61010023_0001_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm China also welcomes general suggestions for additions and improvements.

Note

Arm China tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
NPU driver

This chapter describes the driver of the Zhouyi NPU.

It contains the following sections:

1.1 About the NPU driver

The Zhouyi NPU driver is responsible for parsing middleware generated by an offline NN compiler and scheduling corresponding AI acceleration tasks to the NPU accelerator. Currently, the driver can support NPU controlling on Arm Linux, bare-metal, QNX and RTOS host platforms.

The following figure shows the components of a typical NPU driver.

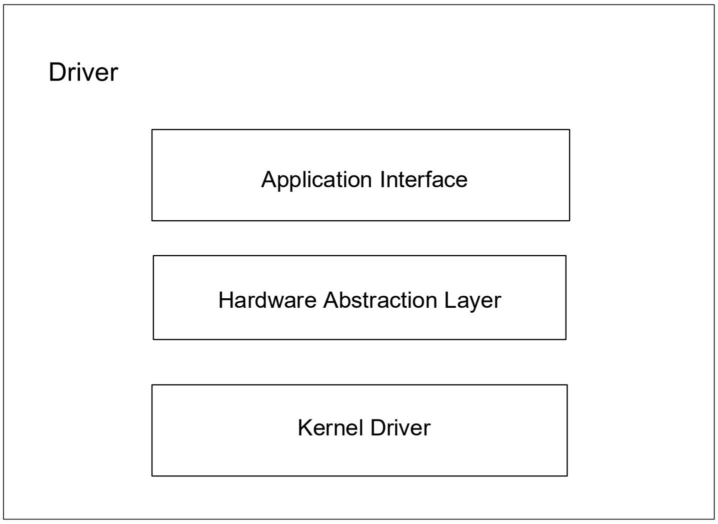

Figure 1-1 Components of a typical NPU driver

1.2 Linux-based driver

The Linux-based driver consists of two key parts—*User Mode Driver* (UMD) and *Kernel Mode Driver* (KMD). The UMD parses NPU job descriptors passed by applications, allocates resources and schedules jobs by calling KMD interfaces.

1.2.1 UMD

The UMD is compiled as a dynamic library and provides user applications a series of interfaces. Standard UMD APIs are designed for applications to schedule standard AI executable benchmarks onto the NPU.

User applications should call the APIs in a relatively fixed order and check the return values step by step to ensure correct operations.

The typical usage of the standard UMD interfaces in an application is as follows:

1. The application calls the context initialization API first to initialize UMD runtime context.
2. After initialization, the application should call the graph loading API to load an offline built binary. If successful, a graph ID is returned. Multiple graph binaries can be loaded in order.
3. After graph loading, the application can create and schedule an inference job which is bound to a previously loaded graph. All input, output, and intermediate buffers are allocated in this step.
4. The application should load the input frame data into the corresponding input buffers.
5. If the blocking scheduling API is used and returns without an error code, the application can get the output tensor directly from the tensor output buffer. If the non-blocking scheduling API is used, after querying the status of execution and no exception is found, the application can fetch the results in the same way.
6. The application should clean a finished job after it terminates, or reuse the job with new input data. The bind buffer can be reused in loading the inputs of the next frame and rescheduling the job again as described in steps 4–5.
7. Before exiting, the application should unload all graphs and destroy the context.

For more information about the corresponding data structures and interfaces, see *5.2.3 Linux driver user interface - Standard API*. Additionally some samples are supplied for reference in driver/samples.

1.2.2 KMD

The KMD works closely with the Linux kernel to respond to contiguous physical memory allocation requests from the UMD, controls job scheduling on the NPU, and handles NPU interrupts. The NPU KMD supports the management of both multiple NPU core instances with heterogeneous configurations (for AIPUV2/V3) and partition-cluster architecture (for AIPU V3). System software developers porting the NPU driver should implement the corresponding Linux device tree to tell the KMD the low-level hardware information. For more information about the device tree bindings and other usage/configuration options of the kernel driver, see the KMD readme document in the release package of driver source code.

To support UMD requests, like other char drivers, several standard Linux device file operation interfaces are provided to the UMD—open, release, mmap, ioctl and poll.

The ioctl options supported are as follows:

- AIPU_IOCTL_QUERY_CAP: Queries the common capability of NPUs.
- AIPU_IOCTL_QUERY_PARTITION_CAP: Queries the capability of an NPU core or partition.
- AIPU_IOCTL_REQ_BUF: Requests to allocate a coherent buffer.

- AIPU_IOCTL_FREE_BUF: Requests to free a coherent buffer allocated by AIPU_IOCTL_REQBUF.
- AIPU_IOCTL_DISABLE_SRAM: Disables the management of SoC SRAM in the kernel driver.
- AIPU_IOCTL_ENABLE_SRAM: Enables the management of SoC SRAM in the kernel driver disabled by AIPU_IOCTL_DISABLE_SRAM.
- AIPU_IOCTL_SCHEDULE_JOB: Schedules a user job to the kernel mode driver for execution.
- AIPU_IOCTL_QUERY_STATUS: Queries the execution status of one or multiple scheduled jobs.
- AIPU_IOCTL_KILL_TIMEOUT_JOB: Kills a timeout job and cleans it from the kernel mode driver.
- AIPU_IOCTL_REQ_IO: Reads/writes an external register of an NPU core.
- AIPU_IOCTL_GET_HW_STATUS: Gets the working status of NPU cores (for example, busy or idle).
- AIPU_IOCTL_ABORT_CMD_POOL: Aborts the NPU command pools (for AIPUv3 only).
- AIPU_IOCTL_DISABLE_TICK_COUNTER: Disables the tick counter (for AIPUv3 only).
- AIPU_IOCTL_ENABLE_TICK_COUNTER: Enables the tick counter (for AIPUv3 only).
- AIPU_IOCTL_CONFIG_CLUSTERS: Configures the NPU clusters, for example, disabling cores (for AIPUv3 only).
- AIPU_IOCTL_ALLOC_DMA_BUF: Requests to allocate a buffer using the dma-buf framework.
- AIPU_IOCTL_FREE_DMA_BUF: Frees a buffer allocated with the dma-buf allocation ioctl.
- AIPU_IOCTL_GET_DMA_BUF_INFO: Gets description of a buffer allocated with the dma-buf allocation ioctl.
- AIPU_IOCTL_GET_DRIVER_VERSION: Gets a string that contains the driver version number.

To port the KMD on a device, refer to the guide `AI610-SDK-1012/Linux-driver/driver/kmd/porting_guide.txt`.

1.2.3 Linux driver user interface - Standard API

Data structure

Data structure	Type	Members	Comment
device_status_t	enum	DEV_IDLE (0x0)	-
		DEV_BUSY	-
		DEV_EXCEPTION	-

Data structure	Type	Values	Comment
aipu_data_type_t	enum	TENSOR_DATA_TYPE_NONE (0x0)	No data type.
		AIPU_DATA_TYPE_BOOL(0x1)	Bool type.
		TENSOR_DATA_TYPE_U8 (0x2)	Unsigned int8.
		TENSOR_DATA_TYPE_S8 (0x3)	Signed int8.
		TENSOR_DATA_TYPE_U16 (0x4)	Unsigned int16.
		TENSOR_DATA_TYPE_S16 (0x5)	Signed int16.
		AIPU_DATA_TYPE_U32 (0x6)	Unsigned int32.
		AIPU_DATA_TYPE_S32 (0x7)	Signed int32.
		AIPU_DATA_TYPE_U64 (0x8)	Unsigned int64.
		AIPU_DATA_TYPE_S64 (0x9)	Signed int64.
		AIPU_DATA_TYPE_f16 (0xA)	Float16.

		AIPU_DATA_TYPE_f32 (0xB)	Float32.
		AIPU_DATA_TYPE_f64 (0xC)	Float64.

Data structure	**Type**	**Members**	**Comment**
aipu_tensor_type_t	enum	AIPU_TENSOR_TYPE_INPUT(0)	-
		AIPU_TENSOR_TYPE_OUTPUT	-
		AIPU_TENSOR_TYPE_INTER_DUMP	-
		AIPU_TENSOR_TYPE_PRINTF	-
		AIPU_TENSOR_TYPE_PROFILER	-
		AIPU_TENSOR_TYPE_LAYER_COUNTER	-
		AIPU_TENSOR_TYPE_ERROR_CODE	-

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_tensor_desc_t	struct	id	uint32_t	Tensor ID.
		size	Uint32_t	Tensor size.
		scale	float	Tensor scale parameter.
		zero_point	float	Tensor zero-point parameter.
		data_type	aipu_data_type_t	Tensor format information.

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_global_config_simulation_t	struct	simulator	const char*	Simulator executable full name.
		x2_arch_desc	const char*	Simulated aipu_v3 ARCH.
		log_file_path	const char*	The file path to store log files.
		log_level	uint32_t	Log level.
		gm_size	uint32_t	GM size for the aipu_v3 simulator
		verbose	bool	Verbose switch.
		enable_avx	bool	Simulator ENABLE_AVX switch.
		enable_calloc	bool	Simulator ENABLE_CALLOC switch.
		en_eval	bool	Simulator ENABLE_EVAL switch.

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_job_config_simulation_t	struct	data_dir	const char*	The data path to store simulation dumped data files.

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_job_config_dump_t	struct	dump_dir	const char*	The data path to store dumped data files.
		prefix	const char *	The file name prefix for dumped files.
		output_prefix	const char*	The file name prefix for dumped output files (can be ignored).

		misc_prefix	const char*	The file name prefix for dumped profile files.

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_create_job_cfg_t	struct	partition_id	uint8_t	Indicates which cluster partition the created job is committed to. Only for aipu_v3.
		dbg_dispatch	uint8_t	Debug dispatch flag. When set to 1, it indicates that the job is assigned to the debug core to run.
		dbg_core_id	uint8_t	Specifies the debug core ID [0, max_core_id in the cluster].
		qos_level	uint8_t	Indicates which QoS level the created job is assigned with (low or high). Only for aipu_v3.
		fm_mem_region	uint8_t	Controls which region the feature map buffer memory is allocated from. By default, it is allocated is from DDR. If configured as AIPU_MEM_REGION_SRAM, it is allocated from SRAM (if available and configured on DTS). If configured as AIPU_MEM_REGION_DTCM, it is allocated from DTCM (if available and configured on DTS and the CPU can access).
		wt_mem_region	uint8_t	Controls which region the weight buffer memory is allocated from. By default, it is allocated from DDR. If configured as AIPU_MEM_REGION_SRAM, it is allocated from SRAM (if available and configured on DTS). If configured as AIPU_MEM_REGION_DTCM, it is allocated from DTCM (if available and configured on DTS and the CPU can access).
		fm_idxes	int32_t *	Specifies feature maps allocated from `fm_mem_region`.
		fm_idxes_cnt	int32_t	The element number in `fm_idxes`.
		wt_idxes	int32_t *	Specifies weights allocated from `wt_mem_region`.
		wt_idxes_cnt	int32_t	The element number in `wt_idxes`.

Data structure	**Type**	**Values**	**Comment**
aipu_job_status_t	enum	AIPU_JOB_STATUS_NO_STATUS (0x0)	-
		AIPU_JOB_STATUS_DONE (0x1)	-
		AIPU_JOB_STATUS_EXCEPTION (0x2)	-

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_debugger_job_info_t	struct	instr_base	uint64_t	Instruction section base address (physical).
		simulation_aipu	void *	NPU simulator object.
		simulation_mem_engine	void *	Simulation mem_engine object.

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_shared_tensor_info_t	struct	type	aipu_tensor_type_t	Type of the shared tensor: input or output.
		tensor_idx	uint32_t	The index number of the shared tensor.
		id	uint64_t	Job ID: on marking one IO buffer as shared.

				Graph ID: on specifying the shared buffer to a new graph. (Ignored for `dma_buf`)
		pa	uint64_t	• On marking the shared buffer: return the buffer address of the marked shared IO buffer. • On specifying the shared buffer: transfer the shared buffer address to a new graph. (Ignored for `dma_buf`)
		dmabuf_fd	int	The file ID of a certain `dma_buf`.
		offset_in_dmabuf	uint32_t	The valid offset in `dma_buf` referenced by `dmabuf_fd`.

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_dmabuf_op	struct	dmabuf_fd	int	The file `fd` of `dma_buf`.
		offset_in_dmabuf	uint32_t	The `dma_buf` referenced by `dmabuf_fd`.
		size	uint32_t	Filled data size or fetched data size.
		data	char *	The buffer of data that is written to `dma_buf` or that is read back from `dma_buf`.

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_core_info_t	struct	umd_version[16]	char	The buffer to store the UMD version number.

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_core_info_t	struct	reg_base	uint64_t	Base address of the NPU core external register.

Data structure	**Type**	**Values**	**Comment**
aipu_ioctl_cmd_t	enum	AIPU_IOCTL_MARK_SHARED_TENSOR	Mark one IO buffer of the job as a shared one.
		AIPU_IOCTL_SET_SHARED_TENSOR	Specify the previously marked shared buffer to replace the IO buffer of the new graph.
		AIPU_IOCTL_SET_PROFILE	Dynamically turn on or off the profiling feature in simulation.
		AIPU_IOCTL_ALLOC_DMABUF	Request one `dma_buf` from KMD.
		AIPU_IOCTL_FREE_DMABUF	Release one `dma_buf` to KMD.
		AIPU_IOCTL_WRITE_DMABUF	Write data to one `dma_buf`.
		AIPU_IOCTL_READ_DMABUF	Read data from one `dma_buf`.
		AIPU_IOCTL_GET_VERSION	Get the version number of UMD and KMD.

Data structure	**Type**	**Values**	**Comment**
aipu_config_type_t	enum	AIPU_JOB_CONFIG_TYPE_DUMP_TEXT	Dump text section(s).

		AIPU_JOB_CONFIG_TYPE_DUMP_WEIGHT	Dump weight section(s).
		AIPU_JOB_CONFIG_TYPE_DUMP_RODATA	Dump rodata section(s).
		AIPU_JOB_CONFIG_TYPE_DUMP_DESCRIPTOR	Dump descriptor section(s).
		AIPU_JOB_CONFIG_TYPE_DUMP_INPUT	Dump input data section(s).
		AIPU_JOB_CONFIG_TYPE_DUMP_OUTPUT	Dump output data tensor(s).
		AIPU_JOB_CONFIG_TYPE_DUMP_TCB_CHAIN	Dump TCB chain tensor(s). Not used for Zhouyi aipu_v1 and aipu_v2.
		AIPU_JOB_CONFIG_TYPE_DUMP_EMULATION	Dump emulation files. Internally used.
		AIPU_CONFIG_TYPE_SIMULATION	Simulation configuration switch.
		AIPU_CONFIG_TYPE_HW	The switch of configuration on hardware.
		AIPU_GLOBAL_CONFIG_TYPE_DISABLE_VER_CHECK	Disable graph bin version check.
		AIPU_GLOBAL_CONFIG_TYPE_ENABLE_VER_CHECK	Enable graph bin version check.

Data structure	Type	Values	Comment
aipu_status_t	enum	AIPU_STATUS_SUCCESS (0x0)	The execution of this API returns successfully without any error.
		AIPU_STATUS_ERROR_NULL_PTR	The arguments passed by user applications to UMD API are NULL pointers which are invalid.
		AIPU_STATUS_ERROR_INVALID_CTX	The context pointer passed by user applications to UMD API is invalid.
		AIPU_STATUS_ERROR_OPEN_DEV_FAIL	UMD fails in opening the device file `/dev/aipu` which may result from the failure in probing the KMD module.
		AIPU_STATUS_ERROR_DEV_ABNORMAL	The NPU device is in abnormal state.
		AIPU_STATUS_ERROR_DEINIT_FAIL	UMD API fails in de-initiating a context.
		AIPU_STATUS_ERROR_UNKNOWN_BIN	The type of the binary loaded is unknown. It cannot be loaded or executed.
		AIPU_STATUS_ERROR_INVALID_CONFIG	The configuration passed by a user application to UMD API is invalid.
		AIPU_STATUS_ERROR_GVERSION_UNSUPPORTED	The version of an executable graph binary passed by a user application is not supported on the current UMD.
		AIPU_STATUS_ERROR_TARGET_NOT_FOUND	UMD fails in finding a matching NPU hardware target for the executable graph passed by a user application to execute.
		AIPU_STATUS_ERROR_INVALID_GBIN	The executable graph binary file passed by a user application to UMD API contains invalid items which cannot be parsed or executed.
		AIPU_STATUS_ERROR_INVALID_GRAPH_ID	Graph ID provided is an invalid one which has been unloaded or never existed.
		AIPU_STATUS_ERROR_OPEN_FILE_FAIL	UMD fails in opening a file passed by a user application.

		AIPU_STATUS_ERROR_MAP_FILE_FAIL	UMD fails in mapping a file passed by a user application.
		AIPU_STATUS_ERROR_READ_FILE_FAIL	UMD fails in reading a file passed by a user application.
		AIPU_STATUS_ERROR_WRITE_FILE_FAIL	UMD fails in writing a file passed by a user application.
		AIPU_STATUS_ERROR_INVALID_JOB_ID	Job ID provided is an invalid one which has been cleaned or never existed.
		AIPU_STATUS_ERROR_JOB_EXCEPTION	The execution of a job ID passed by a user application to UMD API ends with an exception.
		AIPU_STATUS_ERROR_JOB_TIMEOUT	The execution of a job ID passed by a user application to UMD API timed out.
		AIPU_STATUS_ERROR_OP_NOT_SUPPORTED	The operation is not supported in the current UMD version or system environment.
		AIPU_STATUS_ERROR_INVALID_OP	The operation is invalid and not allowed in the current system environment.
		AIPU_STATUS_ERROR_INVALID_SIZE	The size argument is invalid.
		AIPU_STATUS_ERROR_BUF_ALLOC_FAIL	The buffer allocation requests fail because of system memory limitations.
		AIPU_STATUS_ERROR_BUF_FREE_FAIL	The buffer free requests fail because the buffers are in busy state.
		AIPU_STATUS_ERROR_INVALID_CORE_ID	The AIPU core ID that the application provides is invalid and cannot be found in the system.
		AIPU_STATUS_ERROR_RESERVE_SRAM_FAIL	UMD fails in reserving SRAM as the executable binary requested because there is no SoC SRAM or SRAM is busy.
		AIPU_STATUS_ERROR_INVALID_TENSOR_ID	The tensor ID that the application provides is invalid.
		AIPU_STATUS_ERROR_INVALID_CLUSTER_ID	The AIPU cluster ID that the application provides is invalid and cannot be found in the system.
		AIPU_STATUS_ERROR_INVALID_PARTITION_ID	The AIPU partition ID that the application provides is invalid.
		AIPU_STATUS_ERROR_PRINTF_FAIL	UMD fails in parsing the printf buffer and printing corresponding logs.
		AIPU_STATUS_ERROR_INVALID_TENSOR_TYPE	The specified tensor type is invalid for this operation.
		AIPU_STATUS_ERROR_INVALID_GM	The GM is invalid.
		AIPU_STATUS_ERROR_INVALID_SEGMMU	The segmented MMU is invalid.
		AIPU_STATUS_ERROR_INVALID_QOS	The specified QoS level is invalid.
		AIPU_STATUS_ERROR_INVALID_TENSOR_CNT	The specified tensor count is invalid.
		AIPU_STATUS_ERROR_TIMEOUT	Timeout on the polling job's status.
		AIPU_STATUS_ERROR_NO_BATCH_QUEUE	There is no specific batch queue.

		AIPU_STATUS_ERROR_MARK_SHARED_TENSOR	Mark shared tensor: no corresponding tensor.
		AIPU_STATUS_ERROR_SET_SHARED_TENSOR	Set shared tensor: no corresponding tensor.
		AIPU_STATUS_MAX	Maximum error code value.

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_ctx_handle_t	struct	opaque	-	The pointer to this opaque context is returned by the init API.

User application interfaces

Function declaration	aipu_status_t aipu_init_context(aipu_ctx_handle_t** ctx);	
Parameter	ctx	Pointer to a memory location allocated by the application where the UMD stores the opaque context handle struct.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_OPEN_DEV_FAIL AIPU_STATUS_ERROR_DEV_ABNORMAL	
Comments	This API is used to initialize the NPU UMD context.	

Function declaration	aipu_status_t aipu_get_error_message (const aipu_ctx_handle_t* ctx, aipu_status_t status, const char** msg);	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	status	Status returned by UMD standard API.
	msg	Pointer to a memory location allocated by the application where the UMD stores the message string pointer.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX	
Comments	This API is used to query additional information about a status returned by UMD API.	

Function declaration	aipu_status_t aipu_deinit_context(const aipu_ctx_handle_t* ctx);	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_DEINIT_FAIL	
Comments	This API is used to destroy the NPU UMD context.	

Function declaration	aipu_status_t aipu_config_global(const aipu_ctx_handle_t* ctx, uint64_t types, void* config)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	types	Configuration type(s)
	config	Pointer to a memory location allocated by the application where stores the configurations.

Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_CONFIG
Comments	This API is used to configure a specified global option.

Function declaration	aipu_status_t aipu_load_graph(const aipu_ctx_handle_t* ctx, const char* graph, uint64_t* id)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	graph	Pointer to a memory location allocated by the application where stores the graph.
	id	Graph ID returned to the caller.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_OPEN_FILE_FAIL AIPU_STATUS_ERROR_MAP_FILE_FAIL AIPU_STATUS_ERROR_UNKNOWN_BIN AIPU_STATUS_ERROR_GVERSION_UNSUPPORTE AIPU_STATUS_ERROR_TARGET_NOT_FOUND AIPU_STATUS_ERROR_INVALID_GBIN AIPU_STATUS_ERROR_BUF_ALLOC_FAIL AIPU_STATUS_ERROR_RESERVE_SRAM_FAIL AIPU_STATUS_ERROR_INVALID_GM	
Comments	This API loads an offline built NPU executable graph binary from file system.	

Function declaration	aipu_status_t aipu_load_graph_helper(const aipu_ctx_handle_t* ctx, const char* graph_buf, uint32_t graph_size, uint64_t* id)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	graph_buf	Loadable graph binary file data.
	graph_size	The byte size of loadable graph binary data.
	id	Graph ID returned to the caller.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_OPEN_GBIN_FAIL AIPU_STATUS_ERROR_MAP_GBIN_FAIL Other values returned by AIPU_load_graph	
Comments	This API loads a graph from the corresponding data buffer.	

Function declaration	aipu_status_t aipu_unload_graph(const aipu_ctx_handle_t* ctx, uint64_t id);	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	id	Graph ID
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX	

	AIPU_STATUS_ERROR_INVALID_GRAPH_ID
Comments	This API is used to unload a loaded graph.

Function declaration	aipu_status_t aipu_create_job(const aipu_ctx_handle_t* ctx, uint64_t graph, uint64_t* job, aipu_create_job_cfg_t *config = nullptr)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	graph	Graph ID returned by aipu_load_graph.
	job	Pointer to a memory location allocated by the application where the UMD stores the created job ID.
	config	1. Specify the partition ID and QoS level of the job, only for Zhouyi X2. 2. Specify the memory region for the feature map and weight buffer. 3. Bind the job to one core in the cluster. [aipu_v3 only] 4. Specify which weight buffer or feature map buffer to be allocated from the specified weight region or feature map region.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_GRAPH_NOT_EXIST AIPU_STATUS_ERROR_BUF_ALLOC_FAIL	
Comments	This API is used to create a job.	

Function declaration	aipu_status_t aipu_finish_job(const aipu_ctx_handle_t* ctx, uint64_t job_id, int32_t time_out)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	job_id	Job ID returned by aipu_create_job.
	time_out	The timeout parameter for the poll system call.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_JOB_ID AIPU_STATUS_ERROR_INVALID_OP AIPU_STATUS_ERROR_JOB_EXCEPTION AIPU_STATUS_ERROR_JOB_TIMEOUT	
Comments	This API is used to flush a new computation job onto the NPU. This API is a blocking one which will schedule a job and block in waiting for it to be done. The application can safely fetch the inference result data if this API returns successfully.	

Function declaration	aipu_status_t aipu_flush_job(const aipu_ctx_handle_t* ctx, uint64_t job_id, aipu_job_callback_func_t cb_func = nullptr)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	job_id	Job ID returned by aipu_create_job.
	cb_func	Callback function to handle this job. Prototype: int (*aipu_job_callback_func_t)(uint64_t job_id, aipu_job_status_t job_state)

Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_JOB_ID AIPU_STATUS_ERROR_INVALID_OP
Comments	This API is used to flush a new computation job onto the NPU. Unlike aipu_finish_job, this API returns as soon as a job is scheduled. By using this API with multiple tensor buffers' ping-pong operation, the pipeline feature will be enabled. The application should query the job done status before fetching the inference result data. It is optional to assign a callback function to handle the done job timely.

Function declaration	aipu_status_t aipu_get_job_status(const aipu_ctx_handle_t* ctx, uint64_t job_id, aipu_job_status_t* status, int32_t timeout = 0)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	job_id	Job ID returned by aipu_create_job.
	status	Pointer to a memory location allocated by the application where the UMD stores the job status.
	timeout	timeout value (ms) to poll the job's status.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_JOB_ID	
Comments	This API is used to get the execution status of a job scheduled via aipu_flush_job. This API is a blocking one which will block in waiting for the job to be done if it is still running. • timeout > 0: The max polling time window is 'timeout'. • timeout = 0: Non-blocking and return the job's status immediately. • timeout = -1: Blocking until the job is really done or an exception occurs.	

Function declaration	aipu_status_t aipu_clean_job(const aipu_ctx_handle_t* ctx, uint64_t id)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	id	Job ID returned by aipu_create_job.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_JOB_ID	
Comments	This API is used to clean a finished job object in UMD. After this API successfully returns, the job ID immediately becomes invalid and cannot be used again. After this API successfully returns, the buffer handle used in creating this job will immediately be free to be used to create another job with the same graph ID.	

Function declaration	aipu_status_t aipu_get_tensor_count(const aipu_ctx_handle_t* ctx, uint64_t id, aipu_tensor_type_t type, uint32_t* cnt)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	id	Job ID returned by aipu_create_job.

	type	Tensor type.
	cnt	Pointer to a memory location allocated by the application where the UMD stores the count.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_GRAPH_ID	
Comments	This API is used to get the tensor count of specified type.	

Function declaration	aipu_status_t aipu_get_tensor_descriptor(const aipu_ctx_handle_t* ctx, uint64_t id, aipu_tensor_type_t type, uint32_t tensor, aipu_tensor_desc_t* desc)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	id	Job ID returned by aipu_create_job.
	type	Tensor type.
	tensor	Tensor ID.
	desc	Pointer to a memory location allocated by the application where the UMD stores the tensor descriptor.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_GRAPH_ID AIPU_STATUS_ERROR_INVALID_TENSOR_ID	
Comments	This API is used to get the tensor descriptor of specified type.	

Function declaration	aipu_status_t aipu_load_tensor(const aipu_ctx_handle_t* ctx, uint64_t id, uint32_t tensor, const void* data)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	id	Job ID returned by aipu_create_job.
	tensor	Tensor ID.
	data	Data of the input tensor.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_JOB_ID AIPU_STATUS_ERROR_INVALID_TENSOR_ID AIPU_STATUS_ERROR_INVALID_OP	
Comments	This API is used to load the input tensor data.	

Function declaration	aipu_status_t aipu_get_tensor(const aipu_ctx_handle_t* ctx, uint64_t job, aipu_tensor_type_t type, uint32_t tensor, void* buf);	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	id	Job ID returned by aipu_create_job.
	type	Tensor type.

	tensor	Tensor ID.
	buf	Pointer to a memory location allocated by the application where the UMD stores the tensor data.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_JOB_ID AIPU_STATUS_ERROR_INVALID_TENSOR_ID AIPU_STATUS_ERROR_INVALID_OP	
Comments	This API is used to get the tensor data of specified type.	

Function declaration	aipu_status_t aipu_config_job(const aipu_ctx_handle_t* ctx, uint64_t id, uint64_t types, void* config)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	id	Job ID returned by aipu_create_job.
	type	Configuration type(s)
	config	Pointer to a memory location allocated by the application where the application stores the configuration data struct.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_JOB_ID AIPU_STATUS_ERROR_INVALID_CONFIG	
Comments	This API is used to configure a specified option of a job.	

Function declaration	aipu_status_t aipu_specify_iobuf(const aipu_ctx_handle_t* ctx, uint64_t id, aipu_shared_tensor_info_t *shared_buf)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	id	Job ID returned by aipu_create_job.
	shared_buf	Specify one `dma_buf` or `non-dma_buf` as the IO buffer of the job.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_JOB_ID AIPU_STATUS_ERROR_INVALID_TENSOR_ID AIPU_STATUS_ERROR_INVALID_SIZE AIPU_STATUS_ERROR_DMABUF_SHARED_IO	
Comments	This API is used to specify a shared buffer as the input or output buffer of the job. The shared buffer has to be one `dma_buf` currently.	

Function declaration	aipu_status_t aipu_get_partition_count(const aipu_ctx_handle_t* ctx, uint32_t* cnt)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.

	cnt	Pointer to a memory location allocated by the application where the UMD stores the partition count.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_OP	
Comments	This API is used to get the NPU partition count.	

Function declaration	aipu_status_t aipu_get_cluster_count(const aipu_ctx_handle_t* ctx, uint32_t partition_id, uint32_t* cnt)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	partition_id	Indicates which partition it gets the cluster count from. Only one partition currently.
	cnt	Pointer to a memory location allocated by the application where the UMD stores the cluster count.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_OP	
Comments	This API is used to get the NPU cluster count.	

Function declaration	aipu_status_t aipu_get_core_count(const aipu_ctx_handle_t* ctx, uint32_t partition_id, uint32_t cluster, uint32_t* cnt)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	partition_id	Partition ID.
	cluster	Cluster ID.
	cnt	Pointer to a memory location allocated by the application where the UMD stores the core count.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_OP AIPU_STATUS_ERROR_INVALID_CLUSTER_ID	
Comments	This API is used to get the NPU core count in a specific cluster.	

Function declaration	aipu_status_t aipu_debugger_get_core_info(const aipu_ctx_handle_t* ctx, uint32_t core_id, aipu_core_info_t* info)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	core_id	ID of an NPU core.
	info	Pointer to a memory location allocated by the application where the UMD stores the core information.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_CORE_ID	

	AIPU_STATUS_ERROR_DEV_ABNORMAL
Comments	This API is used to get information about an NPU core for the debugger to use.

Function declaration	aipu_status_t aipu_debugger_get_job_info(const aipu_ctx_handle_t* ctx, uint64_t job, aipu_debugger_job_info_t* info)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	job_id	Job ID.
	info	Pointer to a memory location allocated by the application where the UMD stores a pointer to the job information.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_JOB_ID	
Comments	This API is used to get the number of NPU cores in a system for the debugger to use. The NPU core IDs are numbered as [0, cnt - 1), respectively. This API should be called by the debugger before calling any other APIs for the debugger to use.	

Function declaration	aipu_status_t aipu_debugger_bind_job(const aipu_ctx_handle_t* ctx, uint32_t core_id, uint64_t job_id)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	core_id	ID of an NPU core.
	job_id	Job ID returned by aipu_create_job.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_CORE_ID AIPU_STATUS_ERROR_INVALID_JOB_ID AIPU_STATUS_ERROR_INVALID_OP	
Comments	This API binds a created job to an idle NPU core for execution later. External registers of the specified NPU core is written after this API returns, but the start PC register is not triggered to run. For the same core or job, it should only be bound once, unless the job is done. The core to be bound should be in idle state, otherwise UMD returns error code	

Function declaration	aipu_status_t aipu_debugger_run_job(const aipu_ctx_handle_t* ctx, uint32_t job_id);	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	job_id	Job ID returned by aipu_create_job.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_JOB_NOT_EXIST AIPU_STATUS_ERROR_JOB_EXCEPTION AIPU_STATUS_ERROR_INVALID_OP	
Comments	This API triggers a previously bound job to run on a target NPU core. A debugger job must be bound before running.	

	After this API successfully returns, the start PC register of the NPU core previously bound with the job specified by job_id is triggered. This API is a blocking API which returns after the job execution ends on hardware. A debugger job should only be scheduled by aipu_debugger_run_job(). You cannot bind a job by aipu_debugger_bind_job() but schedule it by calling aipu_flush_job() or aipu_finish_job().

Function declaration	aipu_status_t aipu_debugger_malloc(const aipu_ctx_handle_t* ctx, uint32_t size, void** va)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	size	Size of the requested buffer in bytes.
	va	Pointer to a virtual address where stores the base address of the buffer.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_SIZE AIPU_STATUS_ERROR_BUF_ALLOC_FAIL	
Comments	This API allocates a buffer for the NPU debugger to use. This API shall be used after aipu_load_graph and before calling aipu_debugger_bind_job.	

Function declaration	aipu_status_t aipu_debugger_free(const aipu_ctx_handle_t* ctx, void* va)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	va	Virtual address to be freed.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_BUF_FREE_ FAIL	
Comments	This API frees a buffer allocated by aipu_debugger_malloc.	

Function declaration	aipu_status_t aipu_printf(char* printf_base, char* redirect_file)	
Parameter	printf_base	Pointer to a tensor buffer where stores the printf log data.
	redirect_file	Printf output redirect file path.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_PRINTF_FAIL	
Comments	This API prints NPU execution log information after the corresponding job ends. The application can choose to redirect the printf log to the terminal or a file. It passes NULL to the redirect_file results in printing logs to the terminal directly. Nothing will be printed if the printf function is not used in the NPU executable binary loaded by aipu_load_graph.	

Function declaration	aipu_status_t aipu_create_batch_queue(const aipu_ctx_handle_t *ctx, uint64_t graph_id, uint32_t *queue_id)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	graph_id	Graph ID returned by aipu_load_graph.

	queue_id	Pointer to store the batch queue ID.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_GRAPH_ID	
Comments	This API is used to create a batch queue.	

Function declaration	aipu_status_t aipu_clean_batch_queue(const aipu_ctx_handle_t *ctx, uint64_t graph_id, uint32_t queue_id)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	graph_id	Graph ID returned by aipu_load_graph.
	queue_id	Queue ID returned by aipu_create_batch_queue.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_GRAPH_ID	
Comments	This API is used to clean a specific batch queue.	

Function declaration	aipu_status_t aipu_config_batch_dump(const aipu_ctx_handle_t *ctx, uint64_t graph_id, uint32_t queue_id, uint64_t types, aipu_job_config_dump_t *dump_cfg)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	graph_id	Graph ID returned by aipu_load_graph.
	queue_id	Queue ID returned by aipu_create_batch_queue.
	types	Dump options for each batch job.
	dump_cfg	The root path to store dump files of each batch.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_GRAPH_ID AIPU_STATUS_ERROR_NO_BATCH_QUEUE	
Comments	This API is used to perform basic configuration for batch inference on the simulator and hardware.	

Function declaration	aipu_status_t aipu_add_batch(const aipu_ctx_handle_t *ctx, uint64_t graph_id, uint32_t queue_id, char *inputs[], char *outputs[])	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	graph_id	Graph ID returned by aipu_load_graph.
	queue_id	Queue ID returned by aipu_create_batch_queue.
	inputs	Buffer pointers for input tensors.
	outputs	Buffer pointers for output tensors.
Return value	AIPU_STATUS_SUCCESS	

	AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_GRAPH_ID AIPU_STATUS_ERROR_NO_BATCH_QUEUE
Comments	This API is used to add a batch buffer for one frame inference.

Function declaration	aipu_status_t aipu_finish_batch(const aipu_ctx_handle_t *ctx, uint64_t graph_id, uint32_t queue_id, aipu_create_job_cfg_t *create_cfg);	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	graph_id	Graph ID returned by aipu_load_graph.
	queue_id	Queue ID returned by aipu_create_batch_queue.
	create_cfg	Config for all batches in one queue.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_GRAPH_ID AIPU_STATUS_ERROR_NO_BATCH_QUEUE	
Comments	This API is used to run multiple batch inferences.	

Function declaration	aipu_status_t aipu_ioctl(aipu_ctx_handle_t *ctx, uint32_t cmd, void *arg = nullptr);	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_context.
	cmd	AIPU_IOCTL_MARK_SHARED_TENSOR: Mark a shared buffer from a belonged job, arg {aipu_shared_tensor_info_t}. The marked buffer is not freed on destroying a job and directly used by a new job. AIPU_IOCTL_SET_SHARED_TENSOR: Specify a marked shared buffer to a new job, arg {aipu_shared_tensor_info_t}. Marking a shared buffer and setting a shared buffer to a new job must be performed in the same one process context. See the sample in `samples/src/sharebuffer_test`. AIPU_IOCTL_ENABLE_TICK_COUNTER: Enable the performance counter, no arg. AIPU_IOCTL_DISABLE_TICK_COUNTER: Disable the performance counter, no arg. AIPU_IOCTL_CONFIG_CLUSTERS: Configure the number of enabled cores in a cluster, arg {struct aipu_config_clusters}. It is used to disable the clock of some cores to reduce power consumption. AIPU_IOCTL_SET_PROFILE: Dynamically enable or disable the profiling feature of AIPUv3 simulation, arg {1/0}. 1: Enable profiling. 0: Disable profiling. AIPU_IOCTL_ALLOC_DMABUF: Request dma_buf from KMD, arg {struct aipu_dma_buf_request}. aipu_dma_buf_request->bytes: Request size (filled by UMD). aipu_dma_buf_request->fd: fd corresponding to dma_buf (filled by KMD).

		AIPU_IOCTL_FREE_DMABUF: Free a dma_buf with its fd, arg { fd }. Refer to the samples in `samples/src/{dmabuf_mmap_test/dmabuf_vmap_test /dmabuf_dma_test}`.
	arg	Input or output argument.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_DEV_ABNORMAL	
Comments	This API is used to send a specific command to the driver. [aipu_v3 only]	

1.2.4 Linux driver user interface - Python API

Public APIs

Function	NPU()
Return value	Returns an NPU class object.
Comments	This API should be called first of all.

Class NPU methods

Function	aipu_init_context()
Parameter	None.
Return value	An integer.
Comments	This method is used to open a device and initialize the UMD context.

Function	aipu_deinit_context()
Parameter	None.
Return value	An integer.
Comments	This method is used to destroy the UMD context.

Function		aipu_get_error_message(status)
Parameter	status	Status returned by other NPU methods.
Return value		A string.
Comments		This method is used to query additional information about a status returned by NPU methods.

Function	aipu_config_global(types, global_config_simulation)	
Parameter	types	Configuration type.
	global_config_simulation	A dictionary contains the configurations.
Return value	An integer.	
Comments	This method is used to configure a specified global option.	

Function	aipu_load_graph(bin_file)	
Parameter	bin_file	The path of the AIPU bin file.
Return value	A dictionary contains the returned status and graph ID.	
Comments	This method loads an offline built NPU executable graph binary from the file system.	

Function	aipu_load_graph(graph_data, length)	
Parameter	graph_data	Returned value by file.read().
	length	The length of the graph_data.
Return value	A dictionary contains the returned status and graph ID.	
Comments	This method loads an offline built NPU executable graph binary from the buffer.	

Function	aipu_unload_graph(graph_id)	
Parameter	graph_id	Returned value by method aipu_load_graph.
Return value	An integer.	
Comments	This method is used to unload a loaded graph.	

Function	aipu_create_job(graph_id, job_config, fm_idxe, wt_idxe)	
Parameter	graph_id	Returned value by method aipu_load_graph.
	job_config	1. Specify the partition ID and QoS level of the job (only for aipu v3). 2. Specify the memory region for the feature map and weight buffer. 3. Bind the job to one core in the cluster. [aipu_v3 only] 4. Specify which weight buffer or feature map buffer to be allocated from the specified weight region or feature map region.
	fm_idxe	
	wt_idxe	
Return value	A dictionary contains the returned status and job ID.	
Comments	This method is used to create a job.	

Function	aipu_config_job(job_id, type, job_config_dump)	
Parameter	jop_id	Returned value by method aipu_create_job.
	type	Configuration type.
	job_config_dump	A dictionary contains the configuration information.
Return value	A dictionary contains the returned status and job ID.	
Comments	This method is used to create a job.	

Function	aipu_finish_job(job_id, timeout)	
Parameter	job_id	Returned value by method aipu_create_job.
	timeout	The number of milliseconds that the method should block in waiting for the job to complete.
Return value	An integer.	

Comments	This method is used to flush a new computation job onto the NPU. This method is a blocking one which will schedule a job and block in waiting for it to be done. The application can safely fetch the inference result data if this method returns successfully.

Function	aipu_flush_job(job_id, py_cb)	
Parameter	job_id	Returned value by method aipu_create_job.
	py_cb	Callback function to handle this job.
Return value	An integer.	
Comments	This method is used to flush a new computation job onto the NPU. Unlike aipu_finish_job, it returns as soon as a job is scheduled. By using this method with multiple tensor buffers' ping-pong operation, the pipeline feature will be enabled. The application should query the job done status before fetching the inference result data. It is optional to assign a callback function to handle the done job timely.	

Function	aipu_get_job_status(job_id, timeout)	
Parameter	job_id	Returned value by method aipu_create_job.
	timeout	The number of milliseconds that the method should block in waiting for the job status.
Return value	A dictionary contains the return status and job status.	
Comments	This method is used to get the execution status of a job scheduled by aipu_flush_job. This method is a blocking one which will block in waiting for the job to be done if it is still running.	

Function	aipu_clean_job(job_id)	
Parameter	job_id	Returned value by method aipu_create_job.
Return value	An integer.	
Comments	This method is used to clean a finished job object in UMD. After this API successfully returns, the job ID immediately becomes invalid and cannot be used again. After this API successfully returns, the buffer handle used in creating this job will immediately be free to be used to create another job with the same graph ID.	

Function	aipu_get_tensor_count(graph_id, type)	
Parameter	graph_id	Returned value by method aipu_load_graph.
	type	Tensor type.
Return value	A dictionary contains the return status and tensor count.	
Comments	This method is used to get the tensor count of the specified type	

Function	aipu_get_tensor_descriptor(graph_id, type, tensor_id)	
Parameter	graph_id	Returned value by method aipu_load_graph.
	type	Tensor type.
	tensor_id	The sequence number of the specified tensor type.
Return value	A data structure contains the tensor information.	
Comments	This method is used to get the specified tensor descriptor.	

Function	aipu_load_tensor_from_file(job_id, tensor_id, file_path)	
Parameter	job_id	Returned value by method aipu_create_job.
	tensor_id	The sequence number of the input tensor.
	file_path	Input tensor file path.
Return value	An integer.	
Comments	This method is used to load the input tensor data.	

Function	aipu_load_tensor(job_id, tensor_id, data)	
Parameter	job_id	Returned value by method aipu_create_job.
	tensor_id	The sequence number of the input tensor.
	data	Returned value of file.read() or numpy array. Only supports int8 numpy array.
Return value	An integer.	
Comments	This method is used to load the input tensor data.	

Function	aipu_get_tensor(job_id, type, tensor_id)	
Parameter	job_id	Returned value by method aipu_create_job.
	type	Tensor type.
	tensor_id	The sequence number of the specified tensor.
Return value	A dictionary contains the return status and tensor data.	
Comments	This method is used to get specified tensor data. Only supports int8 tensor type.	

Function	aipu_ioctl(cmd, py_arg)	
Parameter	cmd	AIPU_IOCTL_MARK_SHARED_TENSOR: Mark a shared buffer from a belonged job. The marked buffer is not freed on destroying a job and directly used by a new job. AIPU_IOCTL_SET_SHARED_TENSOR: Specify a marked shared buffer to a new job. Marking a shared buffer and setting a shared buffer to a new job must be performed in the same one process context. AIPU_IOCTL_EN_TICK_COUNTER: Enable the performance counter, no `arg`. AIPU_IOCTL_DI_COUNTER: Disable the performance counter, no `arg`. AIPU_IOCTL_SET_PROFILE: Dynamically enable or disable the profiling feature of AIPUv3 simulation. AIPU_IOCTL_ALLOC_DMABUF: Request dma_buf from KMD. AIPU_IOCTL_FREE_DMABUF: Free a dma_buf with its `fd`.
	py_arg	A dictionary contains the args.
Return value	A dictionary contains the return status and `cmd` required data.	

Comments	This method is used to send a specific command to the driver. [aipu_v3 only]

Function	aipu_specify_iobuf(job_id, py_arg)	
Parameter	job_id	Returned value by method aipu_create_job.
	py_arg	Specify one dma_buf or non-dma_buf as the job's IO buffer.
Return value	An integer.	
Comments	This method is used to specify a shared buffer as the job's IO buffer. The shared buffer can be one dma_buf or non-dma_buf.	

1.2.5 Driver features

Adapt the SMMU

Except for the device specific reserved memory, the NPU driver also allows to use the physically discontinuous memory regions with the mapping of Arm SMMU v3. To enable an SMMU for the NPU, the corresponding configurations of SMMU v3, the stream ID for the NPU, and DMA ranges should be configured in the device tree. For detailed configurations, see Section 3.1.4.2 in `driver/kmd/porting_guide.txt`. If correctly configured, the KMD will allocate the virtually contiguous addresses for user threads, and both the UMD and the NPU can use these addresses in the same way as they handle the reserved physically contiguous ones.

Specify memory regions

If you configure the DDR, SRAM and DTCM (Zhouyi X1 only) memory regions for the NPU in the device tree file (dts), you need to specify memory regions. It is important to note that the address range of SRAM should be located at ASID0. Refer to the KMD porting guide for detailed configuration.

When creating a job, you can choose which region the weight buffer or feature map buffer is allocated from. The three macros AIPU_MEM_REGION_DEFAULT, AIPU_MEM_REGION_SRAM, and AIPU_MEM_REGION_DTCM represent the DDR, SRAM and DTCM regions, respectively. You can set them to field `fm_mem_region` or `wt_mem_region` in struct `aipu_create_job_cfg`. If there is not enough free space in the specified region, the allocation logic will fall back in the order of DTCM->SRAM->DDR. Actually, you only need to set the most expected region macro other than multiple macros. Refer to the `benchmark_test` sample.

Share IO buffer via DMA-BUF

The driver already supports sharing input/output buffer among modules through the Linux DMA-BUF framework. To apply this scheme, it needs to confirm that the input/output buffer is non-shared with other feature map buffers. In other words, when generating the model binary with the `aipugb` offline tool, it is required to specify such parameters `--disable_input_buffer_reuse` or `--disable_output_buffer_reuse` to keep the input or output as an independent one. When it allocates one `dma_buf` successfully, this `dma_buf` can be used as the input or output buffer of a certain job by configuring struct `aipu_shared_tensor_info {type, tensor_idx, dmabuf_fd, offset_in_dmabuf }`. Refer to samples `dmabuf_mmap_test`, `dmabuf_vmap_test`, and `dmabuf_dma_test`.

Share IO buffer via common buffer

This scheme is used to share IO buffer among different jobs which are created by the same graph or different graphs. It can only be applied in one process context. For example, the output buffer of the first job is shared with the input buffer of another job, or several jobs share one common input buffer.

Specific utilization steps:

1. Be sure to mark a certain IO buffer as a shared one via macro (AIPU_IOCTL_MARK_SHARED_TENSOR).

2. Replace the IO buffer of another job with the previously marked buffer via macro (AIPU_IOCTL_SET_SHARED_TENSOR). Refer to the `sharebuffer_test` sample.

Pipeline mode

This mode allows the application to flush many jobs to the KMD in advance. This is implemented by calling `aipu_flush_job` other than `aipu_finish_job`, where the latter API is for a blocking mode. The committed jobs are queued in the KMD. After the hardware resource is released, the queued job can be dispatched immediately to the NPU. It can save some schedule cost between jobs. Refer to the `flush_job_test` sample.

Job reclaiming thread

The UMD supports two ways to reclaim the job status. The default is the reclaiming thread that is identical to the committing thread of the job. In other words, the job is polled right after the job is committed via `aipu_flush_job` in the same thread context. To adapt to more flexible application scenarios, actually it can create a specific thread to reclaim the job. This is implemented by specifying `poll_in_commit_thread` as false in struct `aipu_global_config_hw_t` and creating the reclaiming thread to call `aipu_get_job_status` and execute other related logic.

Job callback function

If you want to apply different handling logic for separate jobs, you can specify a custom callback function for each job. The callback function is assigned through the third parameter of `aipu_flush_job`. It will be invoked after the job status is polled successfully by the internal `aipu_get_job_status`.

About readme

To fully understand the whole NPU driver implementation and utilization, it is strongly recommended that you first read the readme files in the driver package. These files provide detailed information about compiling the KMD and UMD, the device tree configuration and usages of samples.

1.2.6 Linux driver programming model – Standard API

The following code samples are just for quick reference. The detailed samples are located in the sample folder of the UMD driver source code package. Using the pipeline, multi_batch, multi thread, multi process, and benchmark can run on multiple cores (if any).

Code 1 shows the init & de-init API calls that Linux applications should follow before scheduling specific inference tasks and after exiting inference on the NPU.

```
/**
 * @file main thread
 */

int main(int argc, char* argv[])
{
    /* ... */
    aipu_status_t status = AIPU_STATUS_SUCCESS;
    aipu_ctx_handle_t *ctx = nullptr;
    const char* status_msg = nullptr;

    status = aipu_init_context(&ctx);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
    }

    /* load graphs and schedule jobs */
```

```

    status = aipu_deinit_context(ctx);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
    }

    /* ... */
}
```

Code 1. Initialization and De-Initialization

Code 2 shows the config API call that Linux applications should follow before scheduling specific inference tasks on the NPU simulator.

```
/**
 * @file main thread
 */

int main(int argc, char* argv[])
{
    /* ... */
    aipu_global_config_simulation_t sim_glb_config;

    /* for aipu_v1/v2 */
/* z2/z3/x1_aipu_simulator optionally */
    sim_glb_config.simulator = "./x1_aipu_simulator";

    sim_glb_config.log_level = 0;

    /* for aipu_v3 */
    sim_glb_config.x2_arch_desc = "X2_1204";

    /* initialize context */

    /* config simulation global options */
    ret = aipu_config_global(ctx, AIPU_CONFIG_TYPE_SIMULATION,
&sim_glb_config);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
    }

    /* load graphs and schedule jobs */

    /* deinit */
    /* ... */
}
```

Code 2. Simulation Configuration

Code 3 shows an example of getting the partition number, cluster number, and core number of the NPU before specifying the job to a specific partition.

```
/**
 * @file main thread
 */
```

```

int main(int argc, char* argv[])
{
    uint32_t part_idx = 0, part_cnt = 0, cluster_cnt = 0, core_cnt = 0;

    /* ... */

   /* initialize context */

   /* config simulation global options */

   /* get partition count */
    ret = aipu_get_partition_count(ctx, &part_cnt);
    if (ret != AIPU_STATUS_SUCCESS)
    {
        aipu_get_error_message(ctx, ret, &msg);
        fprintf(stderr, "aipu_get_partition_count: %s\n", msg);
        return ret;
     }

     for (uint32_t i = 0; i < part_cnt; i++)
     {
        /* get cluster count of a partition */
        ret = aipu_get_cluster_count(ctx, i, &cluster_cnt);
        if (ret != AIPU_STATUS_SUCCESS)
        {
            aipu_get_error_message(ctx, ret, &msg);
            fprintf(stderr, "aipu_get_cluster_count: %s\n", msg);
            return ret;
         }

         for (uint32_t j = 0; j < cluster_cnt; j++)
         {
              /* get core count of a cluster within a partition */
              ret = aipu_get_core_count(ctx, i, j, &core_cnt);
              if (ret != AIPU_STATUS_SUCCESS)
              {
                 aipu_get_error_message(ctx, ret, &msg);
                 fprintf(stderr, "aipu_get_core_count: %s\n", msg);
                 return ret;
              }
              fprintf(stdout, "<part_idx, cluster_idx, core_cnt> = <%u, %u, %u>\n", i, j, core_cnt);
          }
     }

    /* load graphs and schedule jobs */

    /* deinit */
    /* ... */
}
```

Code 3. Getting partition, cluster, and core information

Code 4 shows the graph loading and unloading API calls that Linux applications should follow before scheduling specific inference tasks.

```
/**
 * @file main thread
 */

int main(int argc, char* argv[])
```

```
{
    /* ... */
    uint64 graph_id;
    Const char *graph = "./aipu.bin";

    /* initialize context */

    /* load one or multiple graphs from buffer */
    status = aipu_load_graph(ctx, graph, &graph_id);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
    }

    /* alloc tensor buffer, create and schedule jobs in this single thread */

    /* unload one or multiple graphs */
    status = aipu_unload_graph(ctx, graph_id);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */

    }

    /* deinit */
    /* ... */
}
```

Code 4. Graph(s) Loading

Code 5, code 6, and code 7 show the inference job creating, scheduling and execution status checking model that single thread Linux applications should follow.

```
/**
 * @file main thread
 */

int non_pipeline_scheduling()
{
    uint64_t job_id = 0;
    aipu_create_job_cfg_t config = {0};

    /* load graph and allocate buffers */
    /* load input tensors data */

    create_job_cfg.fm_mem_region = 0;
    create_job_cfg. wt_mem_region = 0;
    /* the below 2 lines just for X2, if run for Z2/Z3/X1, it doesn’t need to set this parameter */
    config.partition_id = 0;
    config.qos_level = 0;

    status = aipu_create_job(ctx, graph_id, &job_id, &config);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_status_msg(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
```

```
    }
    status = aipu_finish_job(ctx, job_id, time_out);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
    }
    else
    {
        /* job results fetching */
    }

    /* ... */

    status = aipu_clean_job(ctx, job_id);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
    }
}

int main(int argc, char* argv[])
{
    /* ... */
    /* initialize */

    ret = non_pipeline_scheduling();
    /* ... */

    /* deinit */
    /* ... */
}
```

Code 5. Non-pipeline Scheduling in Single Thread Application

```
/**
 * @file main thread
 */

int pipeline_scheduling()
{
    aipu_create_job_cfg_t config = {0};

    /* ... */
    int pipe_cnt = 3;
    Uint64_t job_id[pipe_cnt];
    int time_out = 1000;
    aipu_job_status_t job_status = AIPU_JOB_STATUS_NO_STATUS;

    /* load graph and allocate buffers */
    /* load input tensors data */

    /* create multiple jobs and then schedule one by one */
    for (uint32_t i = 0; i < pipe_cnt; i++)
    {
        /* load inputs */

        create_job_cfg.fm_mem_region = 0;
        create_job_cfg. wt_mem_region = 0;
```

```
        /* the below 2 lines just for X2 running, if run for Z2/Z3/X1, it doesn’t need to set this parameter */
        config.partition_id = 0;
        config.qos_level = 0;

        /* create job with corresponding buffer handle */
        ret = aipu_create_job(ctx, graph_id, &job_id[i], &config);
        if (ret != AIPU_STATUS_SUCCESS)
        {
            aipu_get_error_message(ctx, ret, &status_msg);
            fprintf(stderr, "Error: %s\n", status_msg);
            /* error handle or quit */
        }

        ret = aipu_flush_job(ctx, job_id[i]);
        if (ret != AIPU_STATUS_SUCCESS)
        {
            aipu_get_error_message(ctx, ret, &status_msg);
            fprintf(stderr, "Error: %s\n", status_msg);
            /* error handle or quit */
        }
    }
    /* application can do some other tasks if any before getting job results */
    /* get job results */
    for (uint32_t i = 0; i < pipe_cnt; i++)
    {
        ret = aipu_get_job_status(ctx, job_id[i], &job_status);
        if ((ret != AIPU_STATUS_SUCCESS) || (job_status != AIPU_JOB_STATUS_DONE))
        {
            aipu_get_error_message(ctx, ret, &status_msg);
            fprintf(stderr, "Error: %s\n", status_msg);
            /* error handle or quit */
        }

        /* fetch job #i result */

        ret = aipu_clean_job(ctx, job_id[i]);
        if (ret != AIPU_STATUS_SUCCESS)
        {
            aipu_get_error_message(ctx, ret, &status_msg);
            fprintf(stderr, "Error: %s\n", status_msg);
            /* error handle or quit */
        }
    }
}

int main(int argc, char* argv[])
{
    /* ... */
    /* initialize */

    ret = pipeline_scheduling();
    /* ... */

    /* free buffers, unload and deinit */
    /* ... */
}
```

Code 6. Pipeline Scheduling in Single Thread Application

```
/**
```

```
 * @file main thread
 */
#define MAX_BATCH 4

int multi_batch_scheduling()
{
    /* ... */
    uint32_t input_cnt, output_cnt;
    char **input_buf = nullptr, **output_buf[MAX_BATCH];
    uint32_t queue_id = 0;
    uint32_t batch_loop_cnt = 2;
    uint64_t graph_id;
    const char* msg = nullptr;
    vector<aipu_tensor_desc_t> input_desc;
    vector<aipu_tensor_desc_t> output_desc;
    aipu_status_t ret = AIPU_STATUS_SUCCESS;

    /* load graph and allocate buffers */
    /* get input and output count */

    /* found input_buf and output_buf */
    input_buf = new char* [input_cnt];
    for (uint32_t i = 0; i < MAX_BATCH; i++)
        output_buf[i] = new char* [output_cnt];

    for (uint32_t i = 0; i < input_cnt; i++)
        input_buf[i] = opt.inputs[i];

    for (uint32_t k = 0; k < MAX_BATCH; k++)
    {
        for (uint32_t i = 0; i < output_cnt; i++)
        {
            char* output = new char[output_desc[i].size];
            output_data[k].push_back(output);
            output_buf[k][i] = output;
        }
    }

    /* create multi batch queue */
    ret = aipu_create_batch_queue(ctx, graph_id, &queue_id);
    if (ret != AIPU_STATUS_SUCCESS)
    {
        aipu_get_error_message(ctx, ret, &msg);
        fprintf(stderr, "aipu_create_batch_queue: %s\n", msg);
        /* error handle or quit */
    }

    ret = aipu_config_batch_dump(ctx, graph_id, queue_id,
         AIPU_CONFIG_TYPE_SIMULATION, &mem_dump_config);
    if (ret != AIPU_STATUS_SUCCESS)
    {
        aipu_get_error_message(ctx, ret, &msg);
        fprintf(stderr, "aipu_config_batch_dump: %s\n", msg);
        /* error handle or quit */
    }

    for (uint32_t round = 0; round < batch_loop_cnt; round++)
    {
        AIPU_INFO() << "Batch round #" << round;
        for (uint32_t i = 0; i < MAX_BATCH; i++)
        {
            /* add batch */
```

```
            ret = aipu_add_batch(ctx, graph_id, queue_id, input_buf,
                    output_buf[i]);
            if (ret != AIPU_STATUS_SUCCESS)
            {
                aipu_get_error_message(ctx, ret, &msg);
                fprintf(stderr, "aipu_add_batch: %s\n", msg);
                /* error handle or quit */
            }
        }

        ret = aipu_finish_batch(ctx, graph_id, queue_id, &create_job_cfg);
        if (ret != AIPU_STATUS_SUCCESS)
        {
            aipu_get_error_message(ctx, ret, &msg);
            fprintf(stderr, "aipu_finish_batch: %s\n", msg);
            /* error handle or quit */
        }
        /* fetch batch #i result */
    }

    ret = aipu_clean_batch_queue(ctx, graph_id, queue_id);
    if (ret != AIPU_STATUS_SUCCESS)
    {
        aipu_get_error_message(ctx, ret, &msg);
        fprintf(stderr, "aipu_clean_batch_queue: %s\n", msg);
        /* error handle or quit */
    }
}

int main(int argc, char* argv[])
{
    /* ... */
    /* initialize */

    ret = multi_batch_scheduling ();
    /* free buffers, unload and deinit */
      }
```

Code 7. Multi-batch Scheduling in Single Thread Application

Code 8 shows the pipeline scheduling in multithread application.

```
/**
 * @file main thread
 */

int main(int argc, char* argv[])
{

    /* initialize context */
    void *data1 = nullptr, *data2 = nullptr;

    /* initialize data1 & data2 */

    /* create more scheduling threads to load graphs independently and run jobs */
    std::thread t1(job_scheduling_thread， data1);
    std::thread t2(job_scheduling_thread， data2);

    t1.join();
    t2.join();

```

```
    /* deinit */
    /* ... */
}

/**
 * @file scheduling thread
 */

void* job_scheduling_thread(void* data)
{
    /* ... */
    ret = pipeline_scheduling();  /* the same as the scheduling func in Code 6 */
    /* ... */
}
```

Code 8. Pipeline Scheduling in Multithread Application

Code 9 shows the usage of the NPU printf and profiling dump features in a single thread non-pipeline application. In real applications, you can decide whether to use these features. Note that the profiling data generated by the NPU profiler will be ready in the profiling buffer automatically after the profiling job ends. The printf data is also stored to the corresponding buffer if the printf API in the NPU binary is called. Note that currently the printf feature is not supported on aipu_v3.

```
/**
 * @file main thread
 */

int dump_file_helper(const char* fname, const void* src, unsigned int size)
{
    /* write <size> bytes from pointer <src> into a file named <fname> */
}

int main(int argc, char* argv[])
{
    /* ... */
    char fname[4096];
    aipu_status_t status = AIPU_STATUS_SUCCESS;
    const char* status_msg = nullptr;
    aipu_tensor_desc_t desc;
    char *buff = nullptr;

    /* init context and load graphs */
    /* ... */

    /* buffer allocations */
    status = aipu_get_tensor_descriptor(ctx, graph_id, AIPU_TENSOR_TYPE_PRINTF, 0, &desc);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Alloc Error: %s\n", status_msg);
        /* error handling or quitting */
    }
    buff = new char[desc.size];

    /* create, schedule a job and wait for it ends */
    /* ... */

    /* Job Done Now */
    /* Case #9.1: print AIPU log information to terminal */
    status = aipu_get_tensor(m_ctx, job_id, AIPU_TENSOR_TYPE_PRINTF, 0, buff);
```

```
    status = aipu_printf(buff, "print.log");
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Printf Error: %s\n", status_msg);
        /* error handling or quitting */
    }

    /* Case #9.2: get data in profiling-dump buffer(s) */

    /* buffer allocations */
    status = sts = aipu_get_tensor_descriptor(ctx, graph_id, AIPU_TENSOR_TYPE_PROFILER, 0, &desc);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Alloc Error: %s\n", status_msg);
        /* error handling or quitting */
    }
    buff = new char[desc.size];

    status = aipu_get_tensor(m_ctx, job_id, AIPU_TENSOR_TYPE_PROFILER, 0, buff);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Printf Error: %s\n", status_msg);
        /* error handling or quitting */
    }

    /* dump the profiling data generated by profiler as a file */
    /* or do any other operations application would like to */
    snprintf(fname, sizeof(fname), "AIPU_Profiler_Data.bin");
    dump_file_helper(fname, buff, desc.size);


    /* clean jobs and graphs, and deinit */
    /* ... */
}
```

Code 9. Usage of NPU Printf and Profiling Dump Features

Code 10 shows the usage of NPU debugger APIs.

```
/**
 * @file main thread
 */


int main(int argc, char* argv[])
{
    /* ... */
    aipu_status_t status = AIPU_STATUS_SUCCESS;
    const char* status_msg = nullptr;
    uint64_t job_id = 0;
    uint32_t core_cnt = 0;
    aipu_core_info_t* core_info = nullptr;
    aipu_debugger_job_info_t dbg_job_info;

    /* init context, load graphs, allocate buffers */
    /* ... */

```

```
    /* create an AIPU debugger job */
    status = aipu_create_job(ctx, graph_id, &job_id);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Create Job Error: %s\n", status_msg);
        /* error handling or quitting */
    }
    /* The following 5 APIs should be called after aipu_create_job and before AIPU_clean_job */

    status = aipu_get_core_cnt(ctx, 0, 0, &core_cnt);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
    }
    core_info = new aipu_core_info_t[core_cnt];
    for (uint32_t i = 0; i < core_cnt; i++)
    {
        status = aipu_debugger_get_core_info(ctx, i, &core_info[i]);
        if (ret != AIPU_STATUS_SUCCESS)
        {
            aipu_get_error_message(ctx, status, &status_msg);
            fprintf(stderr, "Error: %s\n", status_msg);
            /* error handling or quitting */
        }
    }

    status = aipu_debugger_get_job_info(ctx, job_id, &dbg_job_info);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
    }

    /* bind this job to AIPU core 0 (or another one if applicable) */
    status = aipu_debugger_bind_job(ctx, core_selected, job_id);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
    }

    /* trigger this job to run on the bind core */
    status = aipu_debugger_run_job(ctx, job_id);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
    }

    /* get results and clean all */
}
```

Code 10. Usage of NPU Debugger Features

Code 11 shows the usage of accessing dma_buf via mmap in user mode.

Note that the dma_buf mechanism is mainly for sharing IO buffers across multiple modules or processes. Currently it is only implemented for aipuv1/v2/v3. Refer to the detailed implementation related to the mmap way in the sample `samples/src/dmabuf_mmap_test`. In addition, if you want to use dma_buf in kernel mode via vmap or DMA APIs, refer to the samples in `samples/src/{dmabuf_vmap_test, dmabuf_dma_test}`.

```
#define DEV_EXPORTER "/dev/aipu"
#define DMABUF_SZ 0x100000

int dmabuf_malloc(uint64_t size)
{
    int ret = 0;
    int fd = 0, dmabuf_fd = 0;;
    struct aipu_dma_buf_request dma_buf_req = {0};

    fd = open(DEV_EXPORTER, O_RDWR);
    if (fd < 0)
    {
        ret = -1;
        fprintf(stderr, " open %s [fail]\n", DEV_EXPORTER);
        goto out;
    }

    dma_buf_req.bytes = size;
    ret = ioctl(fd, AIPU_IOCTL_ALLOC_DMA_BUF, &dma_buf_req);
    if (ret < 0)
    {
        fprintf(stderr, "ioctl %s [fail]\n", DEV_EXPORTER);
        goto out;
    }

    dmabuf_fd = dma_buf_req.fd;

out:
    close(fd);
    return dmabuf_fd;
}

int dmabuf_free(int _fd)
{
    int ret = 0;
    int fd = 0;

    fd = open(DEV_EXPORTER, O_RDWR);
    if (fd < 0)
    {
        ret = -1;
        fprintf(stderr, "open %s [fail]\n", DEV_EXPORTER);
        goto out;
    }

    ret = ioctl(fd, AIPU_IOCTL_FREE_DMA_BUF, &_fd);
    if (ret < 0)
    {
        fprintf(stderr, “ioctl %s [fail]\n", DEV_EXPORTER);
        goto out;
    }

out:
    close(fd);
    return ret;
}
```

```

int dmabuf_fill(int fd, char *data, uint32_t size)
{
    int ret = 0;
    char *va = nullptr;

    va = (char *)mmap(NULL, DMABUF_SZ, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    if (va == MAP_FAILED)
    {
        ret = -1;
        fprintf(stderr, mmap dmabuf [fail]\n");
        goto out;
    }

    memcpy(va, data, size);
    munmap(va, DMABUF_SZ);

out:
    return ret;
}

int main(int argc, char* argv[])
{
    /* ... */
    aipu_status_t status = AIPU_STATUS_SUCCESS;
    const char* status_msg = nullptr;
    uint64_t graph_id, job_id;
    aipu_shared_tensor_info_t share_tensor;
    dmabuf_fd = 0;

    /* init context, load graphs, allocate buffers */
    /* ... */

    /* request dma_buf */
    dmabuf_fd = dmabuf_malloc(DMABUF_SZ);

    /* fill dma_buf with input data */
    dmabuf_fill(dmabuf_fd, data, size);

    /*
     * construct share buffer arguments
     * @dmabuf_fd: the fd of dma_buf
     * @offset_in_dmabuf: the start offset of valid data in dma_buf
         * @tensor_idx: the tensor index which the dma_buf will replace
         * @type: the replaced tensor type(input or output)
         */
        share_tensor.dmabuf_fd = dmabuf_fd;
        share_tensor.offset_in_dmabuf = 0;
        share_tensor.tensor_idx = 0;
        share_tensor.type = AIPU_TENSOR_TYPE_INPUT;

        status = aipu_create_job(ctx, graph_id, &job_id);
        if (status != AIPU_STATUS_SUCCESS)
        {
            aipu_get_error_message(ctx, status, &status_msg);
            fprintf(stderr, "Error: %s\n", status_msg);
        }

        /* specify dma_buf as input tesnsor buffer-0 */
        status = aipu_specify_iobuf(ctx, job_id, &share_tensor);
        if (status != AIPU_STATUS_SUCCESS)
```

```
      {
          aipu_get_error_message(ctx, status, &status_msg);
          fprintf(stderr, "Error: %s\n", status_msg);
      }

      /* aipu_finish_job ... */

      /* aipu_clean_job, aipu_deinit_ctx ... */
    }
```

Code 11. Usage of dma_buf via the mmap method in user mode

Code 12 shows the usage of aipu_ioctl to handle some miscellaneous requests with aipu_v3.

The aipu_ioctl API supports many miscellaneous controls for aipu_v3, for example, request or free dma_buf, enable or disable some idle cores in a cluster, and enable or disable the performance counter. In addition, it also can be used to share some IO buffers in the same process context.

```
int main(int argc, char* argv[])
{
    /* ... */
    aipu_status_t status = AIPU_STATUS_SUCCESS;
    const char* status_msg = nullptr;
    aipu_ctx_handle_t ctx;

    status = aipu_init_context(&ctx);

    /* load graphs, allocate buffers */
    /* ... */

    /* demo1: enable performance tick counter */
    status = aipu_ioctl(ctx, AIPU_IOCTL_ENABLE_TICK_COUNTER, nullptr);

    /* demo2: enable performance tick counter
     *
     * Just enable one core in cluster 0
     */
    struct aipu_config_clusters confg_clusters;
    memset(&confg_clusters, 0, sizeof(confg_clusters));
    confg_clusters.clusters[0].en_core_cnt = 1;
    status = aipu_ioctl(ctx, AIPU_IOCTL_CONFIG_CLUSTERS, &confg_clusters);

    /* demo3: alloc/free dma_buf */
    struct aipu_dma_buf_request dma_buf_req = {0};
    dma_buf_req.bytes = 4096; /* request 4KB */
    status = aipu_ioctl(ctx, AIPU_IOCTL_ALLOC_DMABUF, &dma_buf_req);

    /* free dma_buf through its fd */
    status = aipu_ioctl(ctx, AIPU_IOCTL_FREE_DMABUF, dma_buf_req.fd);

    if (status != AIPU_STATUS_SUCCESS)
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
    }

    /* aipu_finish_job ... */
    /* get result ... */

    /* aipu_clean_job, ... */
    status = aipu_init_context(ctx);
    if (status != AIPU_STATUS_SUCCESS)
```

```
    {
        aipu_get_error_message(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
    }
}
```

Code 12. Usage of dma_buf via the mmap method in user mode

1.2.7 Linux driver programming model – Python APIs

Code 13 shows the usage of the NPU Linux driver Python APIs.

```

from libaipudrv import *

# create a npu object and init UMD context.
npu = NPU()
ret = npu.aipu_init_context()
if ret != 0:
    print(“failed”)
    exit(-1)

# load an AIPU executable binary file
retmap = npu.aipu_load_graph(“path/aipu.bin”)
if retmap[“ret”] == 0:
    graph_id = retmap[“data”]
else:
    exit(-1)

# get input tensor count
retmap = npu.aipu_get_tensor_count(graph_id, AIPU_TENSOR_TYPE_INPUT)
if retmap[“ret”] == 0:
    input_cnt = retmap[“data”]
else:
    exit(-1)

# get output tensor count
retmap = npu.aipu_get_tensor_count(graph_id, AIPU_TENSOR_TYPE_OUTPUT)
if retmap[“ret”] == 0:
    output_cnt = retmap[“data”]
else:
    exit(-1)

# create an AIPU job
fm_idxes = []
wt_idxes = []
job_cfg = {“partition_id”:0, “dbg_dispatch”:0,
           “dbg_core_id”:0, “qos_level”:0}
retmap = npu.aipu_create_job(graph_id, job_cfg, fm_idxes, wt_idxes)
if retmap[“ret”] == 0:
    job_id = retmap[“data”]
else:
    exit(-1)

# load input tensor
npu.aipu_load_tensor_from_file(job_id, 0, “path/input0.bin”)

# flush job to AIPU HW and waiting, -1 indicate wait forever
ret = npu.aipu_finish_job(job_id, -1)
if ret != 0:
    exit(-1)

```

```
# get output tensor data
retmap = npu.aipu_get_tensor(job_id, AIPU_TENSOR_TYPE_OUTPUT, 0)
if retmap[“ret”][0] == 0:
    output_data = retmap[“data”]
else:
    exit(-1)

# clean job and unload graph
ret = npu.aipu_clean_job(job_id)
ret = npu.aipu_unload_graph(graph_id)

# destroy UMD context
ret = npu.aipu_deinit_context()

```

Code 13. Usage of NPU Driver Python APIs

1.3 QNX-based driver (for customized solutions only)

The QNX-based driver consists of two key parts—*User Mode Driver* (UMD) and *Resource Manager Driver* (RMD). The UMD parses NPU job descriptors passed by applications, allocates resources and schedules jobs by calling RMD interfaces.

1.3.1 UMD

The UMD is compiled as a dynamic library or static library and provides user applications a series of interfaces. Standard UMD APIs are designed for applications to schedule standard AI executable benchmarks onto the NPU.

User applications should call the APIs in a relatively fixed order and check the return values step by step to ensure correct operations.

The typical usage of the standard UMD interfaces in an application is as follows:

1. The application calls the context initialization API first to initialize UMD runtime context.
2. After initialization, the application should call the graph loading API to load an offline built binary. If successful, a graph ID is returned. Multiple graph binaries can be loaded in order.
3. After graph loading, the application can create and schedule an inference job which is bound to a previously loaded graph. All input, output, and intermediate buffers are allocated in this step.
4. The application should load the input frame data into the corresponding input buffers.
5. If the blocking scheduling API is used and returns without an error code, the application can get the output tensor directly from the tensor output buffer. If the non-blocking scheduling API is used, after querying the status of execution and no exception is found, the application can fetch the results in the same way.
6. The application should clean a finished job after it terminates and then the bind buffer can be reused in loading the next frame's inputs and rescheduling the job again as described in steps 4–5.
7. Before exiting, the application should unload all graphs and destroy the context.

For more information about the corresponding data structures and interfaces, see *5.3.3 QNX driver user interface - Standard API.*

1.3.2 Resource Manager Driver

The Resource Manager Driver is a server program that receives messages from applications. It is running in the user space and communicates with the NPU hardware. When the driver is running, it will generate a device file for each device such as the Linux char devices.

The Resource Manager Driver works as a user space application in QNX to respond to contiguous physical memory allocation requests from the UMD, controls job scheduling on the NPU, and handles NPU interrupts.

To support UMD requests, like other Linux char drivers, several standard file operation interfaces are provided to the UMD—open, close, mmap_device_memory, devctl and poll.

The devctl options supported are as follows:

- AIPU_IOCTL_QUERY_CAP: Queries the common capability of NPUs.
- AIPU_IOCTL_QUERY_PARTITION_CAP: Queries the capability of an NPU core.

- AIPU_IOCTL_REQ_BUF: Requests to allocate a coherent buffer.
- AIPU_IOCTL_FREE_BUF: Requests to free a coherent buffer allocated by AIPU_IOCTL_REQBUF.
- AIPU_IOCTL_SCHEDULE_JOB: Schedules a user job to the kernel mode driver for execution.
- AIPU_IOCTL_QUERY_STATUS: Queries the execution status of one or multiple scheduled jobs.
- AIPU_IOCTL_REQ_IO: Reads/Writes an external register of an NPU core.

1.3.3 QNX driver user interface - Standard API

Data structure

Data structure	Type	Values	Comment
aipu_data_type_t	enum	TENSOR_DATA_TYPE_NONE (0x0)	No data type.
		AIPU_DATA_TYPE_BOOL(0x1)	-
		TENSOR_DATA_TYPE_U8 (0x2)	Unsigned int8.
		TENSOR_DATA_TYPE_S8 (0x3)	Signed int8.
		TENSOR_DATA_TYPE_U16 (0x4)	Unsigned int16.
		TENSOR_DATA_TYPE_S16 (0x5)	Signed int16.
		TENSOR_DATA_TYPE_U32 (0x6)	Unsigned int32.
		TENSOR_DATA_TYPE_S32 (0x7)	Signed int32.
		TENSOR_DATA_TYPE_U64 (0x8)	Unsigned int64.
		TENSOR_DATA_TYPE_S64 (0x9)	Signed int64.
		TENSOR_DATA_TYPE_f16 (0xa)	Float 16.
		TENSOR_DATA_TYPE_f32 (0xb)	Float 32.
		TENSOR_DATA_TYPE_f64 (0xc)	Float 64.

Data structure	Type	Members	Comment
aipu_tensor_type_t	enum	AIPU_TENSOR_TYPE_INPUT	-
		AIPU_TENSOR_TYPE_OUTPUT	-
		AIPU_TENSOR_TYPE_PRINTF	-
		AIPU_TENSOR_TYPE_PROFILER	-
		AIPU_TENSOR_TYPE_INTER_DUMP	-
		AIPU_TENSOR_TYPE_LAYER_COUNTER	-
		AIPU_TENSOR_TYPE_ERROR_CODE	-

Data structure	Type	Members	Member type	Comment
aipu_tensor_desc_t	struct	id	uint32_t	Tensor ID.
		size	Uint32_t	Tensor size.
		scale	float	Tensor scale parameter.
		zero_point	float	Tensor zero-point parameter.

		data_type	aipu_data_type_t	Tensor format information.

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_job_config_dump_t	struct	dump_dir	const char*	The data path to store dumped data files.
		prefix	const char*	The file name prefix for dumped files.
		output_prefix	const char*	The name prefix of output dump files.

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_create_job_cfg_t	struct	misc	unsigned int	-
		partition_id	unsigned char	The default value is 0, that is, in partition-0.
		qos_level	unsigned char	The default value is 0, that is, low priority.

Data structure	**Type**	**Values**	**Comment**
aipu_job_status_t	enum	AIPU_JOB_STATUS_NO_STATUS (0x0)	-
		AIPU_JOB_STATUS_DONE (0x1)	-
		AIPU_JOB_STATUS_EXCEPTION (0x2)	-

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_core_info_t	struct	reg_base	uint64_t	Base address of the NPU core external register.

Data structure	**Type**	**Values**	**Comment**
aipu_config_type_t	enum	AIPU_JOB_CONFIG_TYPE_DUMP_TEXT	Dump text section(s).
		AIPU_JOB_CONFIG_TYPE_DUMP_WEIGHT	Dump weight section(s).
		AIPU_JOB_CONFIG_TYPE_DUMP_RODATA	Dump rodata section(s).
		AIPU_JOB_CONFIG_TYPE_DUMP_DESCRIPTOR	Dump descriptor section(s).
		AIPU_JOB_CONFIG_TYPE_DUMP_INPUT	Dump input data section(s).
		AIPU_JOB_CONFIG_TYPE_DUMP_OUTPUT	Dump output data tensor(s).
		AIPU_JOB_CONFIG_TYPE_DUMP_TCB_CHAIN	Dump TCB chain tensor(s). Not used for aipu v1/v2.
		AIPU_JOB_CONFIG_TYPE_DUMP_EMULATION	Dump emulation files. Internally used.
		AIPU_CONFIG_TYPE_SIMULATION	QNX does not support simulation.
		AIPU_GLOBAL_CONFIG_TYPE_DISABLE_VER_CHECK	Disable graph bin version check.
		AIPU_GLOBAL_CONFIG_TYPE_ENABLE_VER_CHECK	Enable graph bin version check.

Data structure	**Type**	**Values**	**Comment**
aipu_status_t	enum	AIPU_STATUS_SUCCESS (0x0)	The execution of this API returns successfully without any error.

		AIPU_STATUS_ERROR_NULL_PTR	The arguments passed by user applications to UMD API are NULL pointers which are invalid.
		AIPU_STATUS_ERROR_INVALID_CTX	The context pointer passed by user applications to UMD API is invalid.
		AIPU_STATUS_ERROR_OPEN_DEV_FAIL	UMD fails in opening the device file `/dev/aipu` which may result from the failure in probing the KMD module.
		AIPU_STATUS_ERROR_DEV_ABNORMAL	The NPU device is in abnormal state.
		AIPU_STATUS_ERROR_DEINIT_FAIL	UMD API fails in de-initiating a context.
		AIPU_STATUS_ERROR_INVALID_CONFIG	The configuration passed by a user application to UMD API is invalid.
		AIPU_STATUS_ERROR_GVERSION_UNSUPPORTED	The version of an executable graph binary passed by a user application is not supported on the current UMD.
		AIPU_STATUS_ERROR_TARGET_NOT_FOUND	UMD fails in finding a matching NPU hardware target for the executable graph passed by a user application to execute.
		AIPU_STATUS_ERROR_INVALID_GBIN	The executable graph binary file passed by a user application to UMD API contains invalid items which cannot be parsed or executed.
		AIPU_STATUS_ERROR_GRAPH_NOT_EXIST	The graph description pointer passed by a user application to UMD API cannot be found in the current context.
		AIPU_STATUS_ERROR_OPEN_FILE_FAIL	UMD fails in opening a file passed by a user application.
		AIPU_STATUS_ERROR_MAP_FILE_FAIL	UMD fails in mapping a file passed by a user application.
		AIPU_STATUS_ERROR_READ_FILE_FAIL	UMD fails in reading a file passed by a user application.
		AIPU_STATUS_ERROR_WRITE_FILE_FAIL	UMD fails in writing a file passed by a user application.
		AIPU_STATUS_ERROR_JOB_NOT_EXIST	The job ID passed by a user application to UMD API cannot be found in the current context.
		AIPU_STATUS_ERROR_JOB_NOT_SCHED	The operation regarding to the job ID passed by a user application to UMD API cannot be executed because it should be done after the job is scheduled.
		AIPU_STATUS_ERROR_JOB_SCHED	The operation regarding to the job ID passed by a user application to UMD API cannot be executed because it should be done before the job is scheduled.
		AIPU_STATUS_ERROR_JOB_NOT_END	The operation regarding to the job ID passed by a user application to UMD API cannot be executed because it should be done after the job execution terminates.
		AIPU_STATUS_ERROR_JOB_EXCEPTION	The execution of a job ID passed by a user application to UMD API ends with an exception.

		AIPU_STATUS_ERROR_JOB_TIMEOUT	The execution of a job ID passed by a user application to UMD API timed out.
		AIPU_STATUS_ERROR_INVALID_OPTIONS	The options passed by a user application to UMD are invalid and cannot be reflected.
		AIPU_STATUS_ERROR_INVALID_PATH	The path passed by a user application to UMD is invalid and cannot be operated.
		AIPU_STATUS_ERROR_OP_NOT_SUPPORTED	The operation is not supported in the current UMD version or system environment.
		AIPU_STATUS_ERROR_INVALID_OP	The operation is invalid and not allowed in the current system environment.
		AIPU_STATUS_ERROR_INVALID_SIZE	The size argument is invalid.
		AIPU_STATUS_ERROR_INVALID_HANDLE	The buffer handle argument is invalid which cannot be found in the current context.
		AIPU_STATUS_ERROR_BUSY_HANDLE	The buffer handle argument cannot be used in this API because it is in busy state and is used by another job.
		AIPU_STATUS_ERROR_BUF_ALLOC_FAIL	The buffer allocation requests fail because of system memory limitations.
		AIPU_STATUS_ERROR_BUF_FREE_FAIL	The buffer free requests fail because the buffers are in busy state.
		AIPU_STATUS_ERROR_RESERVE_SRAM_FAIL	UMD fails in reserving SRAM as the executable binary requested because there is no SoC SRAM or SRAM is busy.
		AIPU_STATUS_ERROR_NO_BATCH_QUEUE	Transfer an incorrect batch queue ID.
		AIPU_STATUS_MAX	Maximum error code value.

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_ctx_handle_t	struct	opaque	-	The pointer to this opaque context is returned by the init API.

User application interfaces

Function declaration	aipu_status_t aipu_init_ctx(aipu_ctx_handle_t** ctx);	
Parameter	ctx	Pointer to a memory location allocated by the application where the UMD stores the opaque context handle struct.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_OPEN_DEV_FAIL AIPU_STATUS_ERROR_DEV_ABNORMAL	
Comments	This API is used to initialize the NPU UMD context.	

Function declaration	aipu_status_t aipu_get_error_message(const aipu_ctx_handle_t* ctx, aipu_status_t status, const char** msg);	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	status	Status returned by UMD standard API.

	msg	Pointer to a memory location allocated by the application where the UMD stores the message string pointer.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX	
Comments	This API is used to query additional information about a status returned by UMD API.	

Function declaration	aipu_status_t aipu_deinit_ctx(const aipu_ctx_handle_t* ctx);	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_DEINIT_FAIL	
Comments	This API is used to destroy the NPU UMD context.	

Function declaration	aipu_status_t aipu_config_global(const aipu_ctx_handle_t* ctx, uint64_t types, void* config)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	types	Configuration type(s)
	config	Pointer to a memory location allocated by the application where stores the configurations.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX	
Comments	This API is used to configure a specified global option.	

Function declaration	aipu_status_t aipu_load_graph(const aipu_ctx_handle_t* ctx, const char* graph, uint64_t* id)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	graph	Pointer to a memory location allocated by the application where stores the graph.
	id	Graph ID returned to the caller.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_OPEN_FILE_FAIL AIPU_STATUS_ERROR_MAP_FILE_FAIL AIPU_STATUS_ERROR_UNKNOWN_BIN AIPU_STATUS_ERROR_GVERSION_UNSUPPORTE AIPU_STATUS_ERROR_TARGET_NOT_FOUND AIPU_STATUS_ERROR_INVALID_GBIN AIPU_STATUS_ERROR_BUF_ALLOC_FAIL AIPU_STATUS_ERROR_RESERVE_SRAM_FAIL	
Comments	This API loads an offline built NPU executable graph binary from the file system.	

Function declaration	aipu_status_t aipu_load_graph_helper(const aipu_ctx_handle_t* ctx, const char* graph_buf, uint32_t graph_size, uint64_t* id)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	graph_buf	Loadable graph binary file data.
	graph_size	The byte size of loadable graph binary data.
	id	Graph ID returned to the caller.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_OPEN_GBIN_FAIL AIPU_STATUS_ERROR_MAP_GBIN_FAIL Other values returned by AIPU_load_graph	
Comments	This API loads a graph from the corresponding data buffer.	

Function declaration	aipu_status_t aipu_unload_graph(const aipu_ctx_handle_t* ctx, uint64_t id);	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	id	Graph ID.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_GRAPH_ID	
Comments	This API is used to unload a loaded graph.	

Function declaration	aipu_status_t aipu_create_job(const aipu_ctx_handle_t* ctx, uint64_t id, uint64_t* job, aipu_create_job_cfg_t *config)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	id	Pointer to a graph descriptor returned by aipu_load_graph.
	job	Pointer to a memory location allocated by the application where the UMD stores the created job ID.
	config	Specify the partition ID and QoS level of the job, only for AIPUv3.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_GRAPH_NOT_EXIST AIPU_STATUS_ERROR_BUF_ALLOC_FAIL	
Comments	This API is used to create a job.	

Function declaration	aipu_status_t aipu_finish_job(const aipu_ctx_handle_t* ctx, uint64_t job_id, int32_t time_out)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	job_id	Job ID returned by aipu_create_job.
	time_out	The timeout parameter for the poll system call.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR	

	AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_JOB_ID AIPU_STATUS_ERROR_INVALID_OP AIPU_STATUS_ERROR_JOB_EXCEPTION AIPU_STATUS_ERROR_JOB_TIMEOUT
Comments	This API is used to flush a new computation job onto the NPU. This API is a blocking one which will schedule a job and block in waiting for it to be done. The application can safely fetch the inference result data if this API returns successfully.

Function declaration	aipu_status_t aipu_flush_job(const aipu_ctx_handle_t* ctx, uint64_t job_id, void* priv)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	job_id	Job ID returned by aipu_create_job.
	priv	Pointer to the private data structure of the UMD application that is used as an argument of the callback when it is called.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_JOB_ID AIPU_STATUS_ERROR_INVALID_OP	
Comments	This API is used to flush a new computation job onto the NPU. Unlike aipu_finish_job, this API returns as soon as a job is scheduled. By using this API with the ping-pong operation of multiple tensor buffers, the pipeline feature will be enabled. The application should query the job done status before fetching the inference result data.	

Function declaration	aipu_status_t aipu_get_job_status(const aipu_ctx_handle_t* ctx, uint64_t job_id, aipu_job_status_t* status)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	job_id	Job ID returned by aipu_create_job.
	status	Pointer to a memory location allocated by the application where the UMD stores the job status.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_JOB_ID	
Comments	This API is used to get the execution status of a job scheduled via aipu_flush_job. This API is a blocking one which will block in waiting for the job to be done if it is still running.	

Function declaration	aipu_status_t aipu_clean_job(const aipu_ctx_handle_t* ctx, uint64_t id)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	id	Job ID returned by aipu_create_job.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_JOB_ID	

Comments	This API is used to clean a finished job object in UMD. After this API successfully returns, the job ID immediately becomes invalid and cannot be used again. After this API successfully returns, the buffer handle used in creating this job will immediately be free to be used to create another job with the same graph ID.

Function declaration	aipu_status_t aipu_get_tensor_count(const aipu_ctx_handle_t* ctx, uint64_t id, aipu_tensor_type_t type, uint32_t* cnt)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	id	Job ID returned by aipu_create_job.
	type	Tensor type.
	cnt	Pointer to a memory location allocated by the application where the UMD stores the count.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_GRAPH_ID	
Comments	This API is used to get the tensor count of specified type.	

Function declaration	aipu_status_t aipu_get_tensor_descriptor(const aipu_ctx_handle_t* ctx, uint64_t id, aipu_tensor_type_t type, uint32_t tensor, aipu_tensor_desc_t* desc)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	id	Job ID returned by aipu_create_job.
	type	Tensor type.
	tensor	Tensor ID.
	desc	Pointer to a memory location allocated by the application where the UMD stores the tensor descriptor.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_GRAPH_ID AIPU_STATUS_ERROR_INVALID_TENSOR_ID	
Comments	This API is used to get the tensor descriptor of specified type.	

Function declaration	aipu_status_t aipu_load_tensor(const aipu_ctx_handle_t* ctx, uint64_t id, uint32_t tensor, const void* data)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	id	Job ID returned by aipu_create_job.
	tensor	Tensor ID.
	ddata	Data of the input tensor.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_JOB_ID AIPU_STATUS_ERROR_INVALID_TENSOR_ID	

	AIPU_STATUS_ERROR_INVALID_OP
Comments	This API is used to load the input tensor data.

Function declaration	aipu_status_t aipu_get_tensor(const aipu_ctx_handle_t* ctx, uint64_t job, aipu_tensor_type_t type, uint32_t tensor, void* buf);	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	id	Job ID returned by aipu_create_job.
	type	Tensor type.
	tensor	Tensor ID.
	buf	Pointer to a memory location allocated by the application where the UMD stores the tensor data.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_JOB_ID AIPU_STATUS_ERROR_INVALID_TENSOR_ID AIPU_STATUS_ERROR_INVALID_OP	
Comments	This API is used to get the tensor data of specified type.	

Function declaration	aipu_status_t aipu_config_job(const aipu_ctx_handle_t* ctx, uint64_t id, uint64_t types, void* config)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	id	Job ID returned by aipu_create_job.
	type	Configuration type(s).
	config	Pointer to a memory location allocated by the application where the application stores the configuration data struct.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_JOB_ID AIPU_STATUS_ERROR_INVALID_CONFIG	
Comments	This API is used to configure a specified option of a job.	

Function declaration	aipu_status_t aipu_get_partition_count(const aipu_ctx_handle_t* ctx, uint32_t* cnt)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	cnt	Pointer to a memory location where stores the partition count.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_OP	
Comments	This API is used to get AIPU partition count, only for AIPUv3.	

Function declaration	aipu_status_t aipu_get_cluster_count(const aipu_ctx_handle_t* ctx, uint32_t partition_id, uint32_t* cnt)

Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	partition_id	Indicates which partition it gets the cluster count from.
	cnt	Pointer to a memory location where stores the cluster count.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_OP	
Comments	This API is used to get AIPU cluster count, only for AIPUv3.	

Function declaration	aipu_status_t aipu_get_core_count(const aipu_ctx_handle_t* ctx, uint32_t partition_id, uint32_t cluster, uint32_t* cnt)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	partition_id	Partition ID.
	cluster	Cluster ID.
	cnt	Pointer to a memory location where stores the core count.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_OP	
Comments	This API is used to get AIPU core count, only for AIPUv3.	

Function declaration	aipu_status_t aipu_get_target(const aipu_ctx_handle_t *ctx, char *target)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	target	Pointer to a memory where stores the hardware ARCH information.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX	
Comments	This API is used to get NPU ARCH information.	

Function declaration	aipu_status_t aipu_get_device_status(const aipu_ctx_handle_t* ctx, uint32_t *status)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	status	Pointer to a memory where stores the hardware status.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX	
Comments	This API is used to get NPU status.	

Function declaration	aipu_status_t aipu_create_batch_queue(const aipu_ctx_handle_t* ctx, uint64_t graph_id, uint32_t *queue_id)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.

	graph_id	Graph ID.
	queue_id	The batch queue ID will be saved in pointer queue_id.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_GRAPH_ID	
Comments	This API is used to create a batch queue.	

Function declaration	aipu_status_t aipu_clean_batch_queue(const aipu_ctx_handle_t* ctx, uint64_t graph_id, uint32_t queue_id)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	graph_id	Graph ID.
	queue_id	The batch queue ID.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_GRAPH_ID	
Comments	This API is used to clean a specific batch queue.	

Function declaration	aipu_status_t aipu_add_batch(const aipu_ctx_handle_t* ctx, uint64_t graph_id, uint32_t queue_id, char *inputs[], char *outputs[])	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	graph_id	Graph ID.
	queue_id	The batch queue ID.
	inputs	Buffer pointers for input tensors.
	outputs	Buffer pointers for output tensors.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_GRAPH_ID AIPU_STATUS_ERROR_NO_BATCH_QUEUE	
Comments	This API is used to add group buffer of one benchmark to the batch queue.	

Function declaration	aipu_status_t aipu_finish_batch(const aipu_ctx_handle_t* ctx, uint64_t graph_id, uint32_t queue_id, aipu_create_job_cfg_t *create_cfg)	
Parameter	ctx	Pointer to a context handle struct returned by aipu_init_ctx.
	graph_id	Graph ID.
	queue_id	The batch queue ID.
	create_cfg	Configuration for all batches in one queue.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR	

	AIPU_STATUS_ERROR_INVALID_CTX AIPU_STATUS_ERROR_INVALID_GRAPH_ID AIPU_STATUS_ERROR_NO_BATCH_QUEUE
Comments	This API is used to run multiple batches.

Function declaration	aipu_status_t aipu_printf(char* printf_base, char* redirect_file)	
Parameter	printf_base	Pointer to a tensor buffer where stores the printf log data.
	redirect_file	Printf output redirect file path.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_NULL_PTR AIPU_STATUS_ERROR_PRINTF_FAIL	
Comments	This API prints NPU execution log information after the corresponding job ends. The application can choose to redirect the printf log to the terminal or a file. It passes NULL to the redirect_file results in printing logs to the terminal directly. Nothing will be printed if the printf function is not used in the NPU executable binary loaded by aipu_load_graph.	

1.3.4 QNX driver programming model – Standard API

Code 14 shows the init & de-init API calls that QNX applications should follow before scheduling specific inference tasks and after exiting inference on the NPU.

```
/**
 * @file main thread
 */

int main(int argc, char* argv[])
{
    /* ... */
    aipu_status_t status = AIPU_STATUS_SUCCESS;
    aipu_ctx_handle_t *ctx = nullptr;
    const char* status_msg = nullptr;

    status = aipu_init_ctx(&ctx);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_status_msg(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
    }

    /* load graphs and schedule jobs */

    status = aipu_deinit_ctx(ctx);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_status_msg(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
    }

    /* ... */
}
```

Code 14. Initialization and De-Initialization

Code 15 shows the graph loading and unloading API calls that QNX applications should follow before scheduling specific inference tasks.

```
/**
 * @file main thread
 */

int main(int argc, char* argv[])
{
    /* ... */
    aipu_global_config_simulation_t sim_glb_config;


    /* initialize context */

    /* config simulation global options */
    ret = aipu_config_global(ctx, AIPU_CONFIG_TYPE_SIMULATION,
&sim_glb_config);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_status_msg(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
    }

    /* load graphs and schedule jobs */

    /* deinit */
    /* ... */
}
```

Code 15. Graph(s) Loading

Code 16 and 17 show the inference job creating, scheduling and execution status checking model that single thread QNX applications should follow.

```
/**
 * @file main thread
 */

int non_pipeline_scheduling()
{
    uint64_t job_id = 0;
    /* load graph and allocate buffers */
    /* load input tensors data */

    status = aipu_create_job(ctx, graph_id, &job_id);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_status_msg(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
    }
    status = aipu_finish_job(ctx, job_id, time_out);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_status_msg(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
    }
    else
```

```
    {
        /* job results fetching */
    }

    /* ... */

    status = aipu_clean_job(ctx, job_id);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_status_msg(ctx, status, &status_msg);
        fprintf(stderr, "Error: %s\n", status_msg);
        /* error handling or quitting */
    }
}

int main(int argc, char* argv[])
{
    /* ... */
    /* initialize */

    ret = non_pipeline_scheduling();
    /* ... */

    /* deinit */
    /* ... */
}
```

Code 16. Non-pipeline Scheduling in Single Thread Application

```
/**
 * @file main thread
 */

int pipeline_scheduling()
{
    /* ... */
    int pipe_cnt = 3;
    Uint64_t job_id[pipe_cnt];
    int time_out = 1000;
    aipu_job_status_t job_status = AIPU_JOB_STATUS_NO_STATUS;

    /* load graph and allocate buffers */
    /* load input tensors data */

    /* create multiple jobs and then schedule one by one */
    for (uint32_t i = 0; i < pipe_cnt; i++)
    {
        /* load inputs */

        /* create job with corresponding buffer handle */
        ret = aipu_create_job(ctx, graph_id, &job_id[i]);
        if (ret != AIPU_STATUS_SUCCESS)
        {
            aipu_get_status_msg(ctx, ret, &status_msg);
            fprintf(stderr, "Error: %s\n", status_msg);
            /* error handle or quit */
        }

        ret = aipu_flush_job(ctx, job_id[i]);
        if (ret != AIPU_STATUS_SUCCESS)
        {
            aipu_get_status_msg(ctx, ret, &status_msg);
```

```
            fprintf(stderr, "Error: %s\n", status_msg);
            /* error handle or quit */
        }
    }
    /* application can do some other tasks if any before getting job results */
    /* get job results */
    for (uint32_t i = 0; i < pipe_cnt; i++)
    {
        ret = aipu_get_job_status(ctx, job_id[i], &job_status);
        if ((ret != AIPU_STATUS_SUCCESS) || (job_status != AIPU_JOB_STATUS_DONE))
        {
            aipu_get_status_msg(ctx, ret, &status_msg);
            fprintf(stderr, "Error: %s\n", status_msg);
            /* error handle or quit */
        }

        /* fetch job #i result */

        ret = aipu_clean_job(ctx, job_id[i]);
        if (ret != AIPU_STATUS_SUCCESS)
        {
            aipu_get_status_msg(ctx, ret, &status_msg);
            fprintf(stderr, "Error: %s\n", status_msg);
            /* error handle or quit */
        }
    }
}

int main(int argc, char* argv[])
{
    /* ... */
    /* initialize */

    ret = pipeline_scheduling();
    /* ... */

    /* free buffers, unload and deinit */
    /* ... */
}
```

Code 17. Pipeline Scheduling in Single Thread Application

Code 18 shows the pipeline scheduling in multithread application.

```
/**
 * @file main thread
 */

int main(int argc, char* argv[])
{

    /* initialize context */
    void *data1 = nullptr, *data2 = nullptr;

    /* initialize data1 & data2 */

    /* create more scheduling threads to load graphs independently and run jobs */
    std::thread t1(job_scheduling_thread, data1);
    std::thread t2(job_scheduling_thread, data2);
```

```

    t1.join();
    t2.join();

    /* deinit */
    /* ... */
}

/**
 * @file scheduling thread
 */

void* job_scheduling_thread(void* data)
{
    /* ... */
    ret = pipeline_scheduling();  /* the same as the scheduling func in Code 5 */
    /* ... */
}
```

Code 18. Pipeline Scheduling in Multithread Application

Code 19 shows the usage of the NPU printf and profiling dump features in a single thread non-pipeline application. In real applications, you can decide whether to use these features. Note that the profiling data generated by the NPU profiler will be ready in the profiling buffer automatically after the profiling job ends. The printf data is also stored to the corresponding buffer if the printf API in the NPU binary is called.

```
/**
 * @file main thread
 */

int dump_file_helper(const char* fname, const void* src, unsigned int size)
{
    /* write <size> bytes from pointer <src> into a file named <fname> */
}

int main(int argc, char* argv[])
{
    /* ... */
    char fname[4096];
    aipu_status_t status = AIPU_STATUS_SUCCESS;
    const char* status_msg = nullptr;
    aipu_tensor_desc_t desc;
    char *buff = nullptr;

    /* init context and load graphs */
    /* ... */

    /* buffer allocations */
    status = sts = aipu_get_tensor_descriptor(ctx, graph_id, AIPU_TENSOR_TYPE_PRINTF, 0, &desc);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_status_msg(ctx, status, &status_msg);
        fprintf(stderr, "Alloc Error: %s\n", status_msg);
        /* error handling or quitting */
    }
    buff = new char[desc.size];

    /* create, schedule a job and wait for it ends */
    /* ... */
```

```

    /* Job Done Now */
    /* Case #9.1: print AIPU log information to terminal */
    status = aipu_get_tensor(m_ctx, job_id, AIPU_TENSOR_TYPE_PRINTF, 0, buff);
    status = aipu_printf(buff, "print.log");
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_status_msg(ctx, status, &status_msg);
        fprintf(stderr, "Printf Error: %s\n", status_msg);
        /* error handling or quitting */
    }


    /* Case #9.2: get data in profiling-dump buffer(s) */

    /* buffer allocations */
    status = sts = aipu_get_tensor_descriptor(ctx, graph_id, AIPU_TENSOR_TYPE_PROFILER, 0, &desc);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_status_msg(ctx, status, &status_msg);
        fprintf(stderr, "Alloc Error: %s\n", status_msg);
        /* error handling or quitting */
    }
    buff = new char[desc.size];

    status = aipu_get_tensor(m_ctx, job_id, AIPU_TENSOR_TYPE_PROFILER, 0, buff);
    if (AIPU_STATUS_SUCCESS != status)
    {
        aipu_get_status_msg(ctx, status, &status_msg);
        fprintf(stderr, "Printf Error: %s\n", status_msg);
        /* error handling or quitting */
    }

    /* dump the profiling data generated by profiler as a file */
    /* or do any other operations application would like to */
    snprintf(fname, sizeof(fname), "AIPU_Profiler_Data.bin");
    dump_file_helper(fname, buff, desc.size);


    /* clean jobs and graphs, and deinit */
    /* ... */
}
```

Code 19. Usage of NPU Printf and Profiling Dump Features

1.4 Bare-metal-based driver

The bare-metal driver is compiled as a static library, which uses similar user interfaces and workflow as its Linux counterparts, except that no standard operating system APIs are used. The interfaces are listed in *5.4.2 Bare-metal platform programming model*.

User applications should call the APIs in a relatively fixed order and check the return values step by step to ensure correct operations.

The typical usage of bare-metal driver interfaces in an application is as follows:

1. The application calls the address config API first to initialize the NPU External Control Register group base address on the SoC. The memory offset between the SoC and the NPU will also be updated in this API.
2. After the address is set, the application should call the context initialization API to initialize bare-metal runtime context.
3. After context initialization, the application should call the graph loading API to load an offline built binary. If successful, a graph descriptor will be returned. Multiple graph binaries can load in order before next steps.
4. After graph loading, the application should allocate at least one copy of tensor buffers for a graph to be executed. For every graph, the application can allocate multiple copies of tensor buffers to realize executions in the pipeline.
5. According to the graph descriptor information in step 3, the application should allocate the corresponding space for IO tensor descriptors.
6. After IO tensor descriptors are allocated, the application should call the tensor descriptor get API to pass the descriptors to the driver to get specified IO buffer address allocated by the bare-metal runtime driver.
7. The application should load the input frame data into the corresponding buffers obtained in step 6.
8. After the preceding steps finish, the application can call the start API and specify the pipeline index to trigger NPU running.
9. The application should call the NPU status get API to check NPU running status in polling mode. If there are more than one copy of tensor buffers, the application can load the next input into another buffer while the NPU is running.
10. If the application wants to run multiple input frames, it should follow steps 7-9 repeatedly.
11. Before exiting, the application should call the tensor buffer free API, graph unload API and context de-init API in order to free all buffers.

1.4.1 Bare-metal user interface

Data structure

Data structure	Type	Values	Comment
aipu_tensor_layout_t	enum	TENSOR_LAYOUT_NONE (0x0)	-
		TENSOR_LAYOUT_NHWC (0x1)	-
		TENSOR_LAYOUT_NCHW (0x2)	-
		TENSOR_LAYOUT_NWH (0x3)	-
		TENSOR_LAYOUT_NC (0x4)	-

Data structure	Type	Members	Member type	Comment
aipu_tensor_shape_t	struct	N	unsigned int	-
		H		-
		W		-
		C		-

Data structure	Type	Values	Comment
aipu_data_type_t	enum	TENSOR_DATA_TYPE_NONE (0x0)	Unsigned char.
		TENSOR_DATA_TYPE_U8 (0x2)	Unsigned char.
		TENSOR_DATA_TYPE_S8 (0x3)	Signed char.
		TENSOR_DATA_TYPE_U16 (0x4)	Unsigned short.
		TENSOR_DATA_TYPE_S16 (0x5)	Signed short.

Data structure	Type	Members	Member type	Comment
aipu_tensor_fmt_t	struct	layout	aipu_tensor_layout_t	-
		shape	aipu_tensor_shape_t	-
		data_type	aipu_data_type_t	-

Data structure	Type	Values	Comment
aipu_tensor_io_type_t	enum	TENSOR_IO_TYPE_IN(0x0)	-
		TENSOR_IO_TYPE_OUT(0x1)	-
		TENSOR_IO_TYPE_DUMP(0x2)	-

Data structure	Type	Members	Member type	Comment
aipu_tensor_desc_t	struct	fmt	aipu_tensor_fmt_t	-
		io_type	aipu_tensor_io_type_t	-
		buffer_desc	aipu_buffer_desc_t*	-

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_buffer_desc_t	struct	index	unsigned int	Tensor ID.
		size	unsigned int	Tensor size.
		addr	void*	Tensor buffer address.

Data structure	**Type**	**Members**	**Member type**	**Comment**
aipu_graph_desc_t	struct	graph_id	int	Graph ID.
		input_num	unsigned int	Input number of the graph ID.
		output_num	unsigned int	Output number of the graph ID.
		dump_num	unsigned int	Dump number of the graph ID.

Data structure	**Type**	**Values**	**Comment**
aipu_task_status_t	enum	AIPU_TASK_STATUS_NO_STATUS(0x0)	-
		AIPU_TASK_STATUS_RUNNING(0x1)	-
		AIPU_TASK_STATUS_DONE(0x2)	-
		AIPU_TASK_STATUS_EXCEPTION(0x3)	-

Data structure	**Type**	**Values**	**Comment**
aipu_status_t	enum	AIPU_STATUS_SUCCESS	-
		AIPU_STATUS_ERROR	-
		AIPU_STATUS_ERROR_INVALID_ARGS	-
		AIPU_STATUS_ERROR_GRAPH_NOT_EXIST	-
		AIPU_STATUS_ERROR_GRAPH_INIT_FAIL	-
		AIPU_STATUS_ERROR_INVALID_BIN	-
		AIPU_STATUS_ERROR_INVALID_BIN_HEAD_SIZE	-
		AIPU_STATUS_ERROR_INVALID_BIN_MAGIC	-
		AIPU_STATUS_ERROR_PARSE_BSS_STATIC_FAIL	-
		AIPU_STATUS_ERROR_PARSE_BSS_REUSE_FAIL	-
		AIPU_STATUS_ERROR_BUF_ALLOC_FAIL	-
		AIPU_STATUS_ERROR_INVALID_BUFFER	-
		AIPU_STATUS_ERROR_INVALID_GRAPH_VERSION	-
		AIPU_STATUS_ERROR_INIT_CONTEXT_FAIL	-
		AIPU_STATUS_ERROR_DEINIT_CONTEXT_FAIL	-
		AIPU_STATUS_ERROR_CONTEXT_NOT_INIT	-
		AIPU_STATUS_ERROR_TENSOR_NOT_ALLOC	-
		AIPU_STATUS_ERROR_TENSOR_NOT_EXIST	-
		AIPU_STATUS_ERROR_RUN_BUSY	-
		AIPU_STATUS_ERROR_RUN_EXCEPTION	-

		AIPU_STATUS_ERROR_RUN_SUCCESS_BUT_WARNING	-
		AIPU_STATUS_ERROR_GRAPH_IS_RUNNING	-
		AIPU_STATUS_ERROR_GRAPH_ID_NOT_RUN	-
		AIPU_STATUS_ERROR_GRAPH_NOT_MATCH_HARDWARE	-
		AIPU_STATUS_ERROR_PARSE_BSS_LINKED_FAIL	-
		AIPU_STATUS_ERROR_INVALID_IO_HANDLE	-
		AIPU_STATUS_MAX	-

User application interfaces

Function declaration	void aipu_config_address(unsigned long ctrl_reg_base_addr, unsigned long memory_addr_offset);	
Parameter	ctrl_reg_base_addr	External control register base address.
	memory_addr_offset	Memory offset between the CPU and NPU.
Return value	void	
Comments	This API is used to set address information between the CPU and NPU.	

Function declaration	aipu_status_t aipu_init_ctx(void);
Parameter	void
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_INIT_CONTEXT_FAIL
Comments	This API is used to initialize the NPU context.

Function declaration	aipu_status_t aipu_deinit_ctx(void);
Parameter	void
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_DEINIT_CONTEXT_FAIL
Comments	This API is used to destroy the NPU context.

Function declaration	aipu_status_t aipu_load_graph(void *graph, aipu_graph_desc_t *graph_desc);	
Parameter	graph	Graph binary buffer pointer.
	graph_desc	Graph descriptor.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_CONTEXT_NOT_INIT AIPU_STATUS_ERROR_BUF_ALLOC_FAIL AIPU_STATUS_ERROR_GRAPH_INIT_FAIL AIPU_STATUS_ERROR_INVALID_ARGS AIPU_STATUS_ERROR_INVALID_BIN_MAGIC AIPU_STATUS_ERROR_INVALID_BIN_HEAD_SIZE AIPU_STATUS_ERROR_INVALID_GRAPH_VERSION AIPU_STATUS_ERROR_GRAPH_NOT_MATCH_HARDWARE AIPU_STATUS_ERROR_PARSE_BSS_STATIC_FAIL AIPU_STATUS_ERROR_PARSE_BSS_REUSE_FAIL	

	AIPU_STATUS_ERROR_PARSE_BSS_LINKED_FAIL	
Comments	This API is used to load a graph binary for the driver to parse and allocate static buffers.	

Function declaration	aipu_status_t aipu_unload_graph(int graph_id);	
Parameter	graph_id	Graph ID.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_CONTEXT_NOT_INIT AIPU_STATUS_ERROR_GRAPH_NOT_EXIST AIPU_STATUS_ERROR_INVALID_ARGS AIPU_STATUS_ERROR_GRAPH_IS_RUNNING	
Comments	This API is used to unload a loaded graph.	

Function declaration	aipu_status_t aipu_alloc_tensor_buffer(int graph_id, int io_buff_num);	
Parameter	graph_id	Graph ID.
	io_buff_num	Copies of tensor buffers.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_CONTEXT_NOT_INIT AIPU_STATUS_ERROR_GRAPH_NOT_EXIST AIPU_STATUS_ERROR_INVALID_ARGS AIPU_STATUS_ERROR_INVALID_BUFFER AIPU_STATUS_ERRO_BUF_ALLOC_FAIL	
Comments	This API is used to allocate buffers for all input and output tensors of a graph. The io_buff_num must be >=1, which indicates that how many copies of tensor buffers are allocated by runtime. After the API returns successfully, the valid tensor buffer handle is >=0 and < io_buff_num. If the io_buff_num is set to 1, the pipeline feature will be disabled.	

Function declaration	aipu_status_t aipu_get_tensor_desc(int graph_id, int io_buff_handle, aipu_tensor_io_type_t io_type, aipu_tensor_desc_t* tdesc);	
Parameter	graph_id	Graph ID.
	io_buff_handle	The valid tensor buffer handle which is >=0 and < io_buff_num.
	io_type	Tensor IO type.
	tdesc	Tensor descriptor.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_CONTEXT_NOT_INIT AIPU_STATUS_ERROR_GRAPH_NOT_EXIST AIPU_STATUS_ERROR_TENSOR_NOT_ALLOC AIPU_STATUS_ERROR_INVALID_IO_HANDLE AIPU_STATUS_ERROR_TENSOR_INVALID_ARGS	
Comments	This API is used to get input/output/dump tensor descriptor of a graph. Applications load inputs and get outputs through these descriptors.	

Function declaration	aipu_status_t aipu_start(int graph_id , int io_buff_handle);	

Parameter	graph_id	Graph ID.
	io_buff_handle	Valid buffer handle which has loaded inputs for aipu_v1v2. It is useless for aipu_v3, and recommended to use value 0.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_CONTEXT_NOT_INIT AIPU_STATUS_ERROR_GRAPH_NOT_EXIST AIPU_STATUS_ERROR_TENSOR_NOT_ALLOC AIPU_STATUS_ERROR_INVALID_IO_HANDLE AIPU_STATUS_ERROR_RUN_BUSY AIPU_STATUS_ERROR_RUN_EXCEPTION AIPU_STATUS_ERROR_RUN_SUCCESS_BUT_WARNING	
Comments	• Before calling this API, the aipu_alloc_tensor_buffer() and tensor data should be put into tdesc. • aipu_start() and aipu_get_status() should be one-to-one, otherwise, the status will be covered by the next running graph. • If this API returns the exception state, the application can call the aipu_get_status() API to clear the exception status.	

Function declaration	aipu_status_t aipu_get_status(int graph_id, aipu_task_status_t *status);	
Parameter	graph_id	Graph ID.
	status	NPU task status.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_CONTEXT_NOT_INIT AIPU_STATUS_ERROR_GRAPH_NOT_EXIST AIPU_STATUS_ERROR_GRAPH_ID_NOT_RUN AIPU_STATUS_ERROR_RUN_EXCEPTION	
Comments	• This API is non-blocking, so the application needs to poll this API to get status. • When this API finds that the NPU runs into exception state, the API will clear the exception state, and then return the exception state value.	

Function declaration	aipu_status_t aipu_free_tensor_buffer(int graph_id);	
Parameter	graph_id	Graph ID.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_CONTEXT_NOT_INIT AIPU_STATUS_ERROR_GRAPH_NOT_EXIST	
Comments	This API is used to free the tensor buffer.	

1.4.2 Bare-metal platform programming model

Code 20 shows the NPU address configuration API usage. The API should be called before any operation. At the same time, the default values of `AIPU_CTRL_REG_BASE_ADDR` and `MEMORY_ADDR_OFFSET` are set as (`0x64000000u`, `0x80000000u`) in the bare-metal runtime driver.

```
#define AIPU_CTRL_REG_BASE_ADDR (0x64000000u)
#define MEMORY_ADDR_OFFSET  (0x80000000u)

aipu_config_address(AIPU_CTRL_REG_BASE_ADDR, MEMORY_ADDR_OFFSET);
```

```

/* ... */
```

Code 20. NPU Driver Library Use Case – for Bare-metal (Part 1)

Code 21 shows the context init & de-init API call. The init API should be called after `aipu_config_address()` to create an environment for next APIs. The de-init API should be called when the application tries exit on the NPU.

```
aipu_status_t ret = AIPU_STATUS_SUCCESS;

ret = aipu_init_ctx();
if(AIPU_STATUS_SUCCESS != ret){
    printf("aipu context init fail\n");
    /* error handling or quitting */
}

/* ... */

ret = aipu_deinit_ctx();
if (AIPU_STATUS_SUCCESS != ret)
{
    printf("aipu context deinit fail\n");
    /* error handling or quitting */
}

/* ... */
```

Code 21. NPU Driver Library Use Case – for Bare-metal (Part 2)

Code 22 shows the graph loading and unloading. After the loading graph is called, the application can get the graph ID and tensor number information. If the application will not use the graph, it can call the unloading API. Before the unloading graph API is called, the application should call the tensor free API to free tensor buffer allocated by the driver at first.

```
aipu_graph_desc_t graph_desc = {0};

/* config address of cpu and aipu */
/* initialize context */

/* graph_addr is pointer of graph binary */
ret = aipu_load_graph(graph_addr, &graph_desc);
if(AIPU_STATUS_SUCCESS != ret){
    printf("aipu load graph fail\n");
    /* error handling or quitting */
}
/* continue loading if multigraphs used */
/* ... */

/* unload graph */
ret = aipu_unload_graph(graph_desc.graph_id);
if (AIPU_STATUS_SUCCESS != ret)
{
    printf("aipu unload graph fail\n");
    /* error handling or quitting */
}

/* ... */
```

Code 22. NPU Driver Library Use Case – for Bare-metal (Part 3)

Code 23 shows the memory allocation and free API. After loading graph, the application can call the tensor allocate buffer API which supports one or more copies of tensor buffers allocated. These buffers will provide the space for the application to load inputs and get outputs.

```
/* config address of cpu and aipu */
/* initialize context */
/* load graph */

/* alloc tensor buffer, only one copy of buffer, no use of pipeline */
ret = aipu_alloc_tensor_buffer(graph_desc.graph_id, 1);
if(AIPU_STATUS_SUCCESS != ret){
    printf("aipu alloc tensor buffer fail\n");
    /* error handling or quitting */
}

/* alloc tensor descriptor buffer */
/* get tensor descriptor entity from driver */
/* load inputs */
/* start aipu and get status*/
/* ... */

/* free all tensor buffer */
ret = aipu_free_tensor_buffer(graph_desc.graph_id);
if (AIPU_STATUS_SUCCESS != ret)
{
    printf("aipu free tensor buffer fail\n");
    /* error handling or quitting */
}

/* ... */
```

Code 23. NPU Driver Library Use Case – for Bare-metal (Part 4)

Code 24 shows the entity getting API of descriptors. Before calling this API, the application should create input and output tensor descriptor entities, and pass them to the driver level through the following API.

```
/* config address of cpu and aipu */
/* initialize context */
/* load graph */
/* alloc tensor buffer */
/* alloc tensor descriptor buffer */

/* get input tensor descriptor entity through the io buffer handle */
ret = aipu_get_tensor_desc(graph_desc.graph_id, 0, TENSOR_IO_TYPE_IN, &input_tensor_desc);
if(AIPU_STATUS_SUCCESS != ret){
    printf("aipu get tensor desc fail\n");
    /* error handling or quitting */
}

/* get output tensor descriptor entity through the io buffer handle */
ret = aipu_get_tensor_desc(graph_desc.graph_id,0, TENSOR_IO_TYPE_OUT, &output_tensor_desc);
if(AIPU_STATUS_SUCCESS != ret){
    printf("aipu get tensor desc fail\n");
    /* error handling or quitting */
}
/* load inputs */
/* start aipu and get status */
/* ... */
```

Code 24. NPU Driver Library Use Case – for Bare-metal (Part 5)

Code 25 shows the action of loading inputs. `load_inputs` should be realized by the application itself. It is a process of copying input frames from the application layer to the driver layer.

```
/* config address of cpu and aipu */
/* initialize context */
/* load graph */
/* alloc tensor buffer */
/* alloc tensor descriptor buffer */
/** <load_inputs> is aim to copy input_src buffer to input tensor  buffer allocated by runtime driver
   <input_src> is pointer of source input frame buffer
*/
if(1 == graph_desc.input_num){
    ret = load_inputs(&input_tensor_desc.buffer_desc[0],  input0_src);
    if(AIPU_STATUS_SUCCESS != ret){
        printf("load input fail\n");
        /* error handling or quitting */
    }
}
else if(2 == graph_desc.input_num){
    ret = load_inputs(&input_tensor_desc.buffer_desc[0],  input0_src);
    if(AIPU_STATUS_SUCCESS != ret){
        printf("load input fail\n");
        /* error handling or quitting */
    }
    ret = load_inputs(&input_tensor_desc.buffer_desc[1], input1_src);
    if(AIPU_STATUS_SUCCESS != ret){
        printf("load input fail\n");
        /* error handling or quitting */
    }
}
/* start aipu and get status */
/* ... */
```

Code 25. NPU Driver Library Use Case – for Bare-metal (Part 6)

Code 26 shows the NPU start & status get API. The application can get status in polling mode.

```
/* config address of cpu and aipu */
/* initialize context */
/* load graph */
/* alloc tensor buffer */
/* alloc tensor descriptor */
/* load inputs */
/* start aipu and specify which tensor buffer aipu uses */
ret = aipu_start(graph_desc.graph_id, 0);
if(!((AIPU_STATUS_SUCCESS == ret) ||  (AIPU_STATUS_ERROR_RUN_SUCCESS_BUT_WARNING == ret))){
    printf("aipu start fail\n");
    /* error handling or quitting */
}
/* check aipu running status, while(1) just for an example */
while(1){
    aipu_get_status(graph_desc.graph_id, &aipu_status);
    if(AIPU_TASK_STATUS_RUNNING != aipu_status){
        break;
    }
}
/* ... */
```

Code 26. NPU Driver Library Use Case – for Bare-metal (Part 7)

Code 27 shows how to use the pipeline feature. The application will apply two copies of tensor buffers, and complete the ping-pong operation. The get_next_ready_pipeline and get_next_idle_pipeline should be realized by the application itself. They are in charge of managing the pipeline and recording which pipeline is idle or ready.

```
/* config address of cpu and aipu */
/* initialize context */
/* load graph */
/* alloc two copies of tensor buffer */
aipu_alloc_tensor_buffer(graph_desc.graph_id, 0);

/* alloc tensor descriptor buffer */
/* ... */
/* get input tensor descriptor entity through the io buffer handle one by one */
ret = aipu_get_tensor_desc(graph_desc.graph_id,0, TENSOR_IO_TYPE_IN, &input_tensor_desc0);

ret = aipu_get_tensor_desc(graph_desc.graph_id,1, TENSOR_IO_TYPE_IN, &input_tensor_desc1);

while(1){
    /* get the ready pipeline, if none, load input */
    pipeline = get_next_ready_pipeine();
    /* no ready pipeline, load input into it */
    /* load input */
    /* ... */

    /* start aipu and specify which tensor buffer aipu uses */
    aipu_start(graph_desc.graph_id, pipeline);

    while(1){
        aipu_get_status(graph_desc.graph_id, &aipu_status);
        /* when aipu is ongiong,  load input to an idle pipeline */
        if(AIPU_TASK_STATUS_RUNNING == aipu_status){
            index = get_next_idle_pipeline();
            /* load input */
            /* ... */
        }
    }
}
/* free tensor buffer */
/* unload graph */
/* context deinit */

```

Code 27. NPU Driver Library Use Case – for Bare-metal (Part 8)

1.5 RTOS-based driver (for customized solutions only)

The RTOS driver is also compiled as a static library, which keeps the same user interfaces and workflow as the bare-metal driver. Therefore, all programming models in bare metal can be used in RTOS. The RTOS driver provided by Arm China is implemented based on FreeRTOS v10.1.1, but it is easy to be ported to other platforms, like RTX, QNX and RT-Thread.

User applications should call the APIs in a relatively fixed order and check the return values step by step to ensure correct operations.

The typical usage of RTOS driver interfaces in an application is as follows:

1. The application calls the address config API first to initialize the NPU External Control Register group base address on the SoC. The memory offset between the SoC and the NPU will also be updated in this API.
2. After the address is set, the application should call the context initialization API to initialize RTOS runtime context.
3. After context initialization, the application should call the graph loading API to load an offline built binary. If successful, a graph descriptor will be returned. Multiple graph binaries can load in order before next steps.
4. After graph loading, the application should allocate at least one copy of tensor buffers for a graph to be executed. For every graph, the application can allocate multiple copies of tensor buffers to realize executions in the pipeline.
5. According to the graph descriptor information in step 3, the application should allocate the corresponding space for IO tensor descriptors.
6. After IO tensor descriptors are allocated, the application should call the tensor descriptor get API to pass the descriptors to the driver to get specified IO buffer address allocated by the RTOS runtime driver.
7. The application should load the input frame data into the corresponding buffers obtained in step 6.
8. After the preceding steps finish, the application can call the start API and specify the pipeline index to trigger NPU running.
9. The application should call the NPU status get API to check NPU running status in polling mode. If there are more than one copy of tensor buffers, the application can load the next input into another buffer while the NPU is running.
10. If the application wants to run multiple input frames, it should follow steps 7-9 repeatedly.
11. Before exiting, the application should call the tensor buffer free API, graph unload API and context de-init API in order to free all buffers.

1.5.1 FreeRTOS demo

Arm China provides a RTOS demo based on FreeRTOS v10.1.1 in `AI610-SDK-1015-xxxx-xxx/FreeRTOS-driver/demo/`. It is compiled into an executable OS ELF file, which consists of the FreeRTOS lib, the AIPU bare-metal driver and the NPU application.

The typical usage of the demo is as follows:

1. Build the FreeRTOS kernel lib. The lib must include basic APIs of FreeRTOS, such as the memory management component and task scheduler component.
2. According to different environments, choose the file system component, the Ethernet component and other components. They are determined by the users and development boards. It is not a mandatory option.
3. Put the compiled lib into the `bsp/lib/` folder and put the FreeRTOS OS related head files into the `bsp/include/` folder.
4. Create the main() function, which will be called by the FreeRTOS OS to set up an NPU application (for example, run a neural network model). In this main() function, create a task (using FreeRTOS API xTaskCreate()) to call the bare-metal driver APIs to set up an application. For more information about the FreeRTOS APIs, see the FreeRTOS documentation or source code.

Currently the driver supports a single task only.

1.5.2 RTOS user interface

Data structure

Data structure	Type	Values	Comment
aipu_tensor_layout_t	enum	TENSOR_LAYOUT_NONE (0x0)	-
		TENSOR_LAYOUT_NHWC (0x1)	-
		TENSOR_LAYOUT_NCHW (0x2)	-
		TENSOR_LAYOUT_NWH (0x3)	-
		TENSOR_LAYOUT_NC (0x4)	-

Data structure	Type	Members	Member type	Comment
aipu_tensor_shape_t	struct	N	unsigned int	-
		H		-
		W		-
		C		-

Data structure	Type	Values	Comment
aipu_data_type_t	enum	TENSOR_DATA_TYPE_NONE (0x0)	Unsigned char.
		TENSOR_DATA_TYPE_U8 (0x2)	Unsigned char.
		TENSOR_DATA_TYPE_S8 (0x3)	Signed char.
		TENSOR_DATA_TYPE_U16 (0x4)	Unsigned short.
		TENSOR_DATA_TYPE_S16 (0x5)	Signed short.

Data structure	Type	Members	Member type	Comment
aipu_tensor_fmt_t	struct	layout	aipu_tensor_layout_t	-
		shape	aipu_tensor_shape_t	-
		data_type	aipu_data_type_t	-

Data structure	Type	Values	Comment
aipu_tensor_io_type_t	enum	TENSOR_IO_TYPE_IN(0x0)	-
		TENSOR_IO_TYPE_OUT(0x1)	-
		TENSOR_IO_TYPE_DUMP(0x2)	-

Data structure	Type	Members	Member type	Comment
aipu_tensor_desc_t	struct	fmt	aipu_tensor_fmt_t	-
		io_type	aipu_tensor_io_type_t	-
		buffer_desc	aipu_buffer_desc_t*	-

Data structure	Type	Members	Member type	Comment
aipu_buffer_desc_t	struct	index	unsigned int	Tensor ID.
		size	unsigned int	Tensor size.
		addr	void*	Tensor buffer address.

Data structure	Type	Members	Member type	Comment
aipu_graph_desc_t	struct	graph_id	int	Graph ID.
		input_num	unsigned int	Input number of the graph ID.
		output_num	unsigned int	Output number of the graph ID.
		dump_num	unsigned int	Dump number of the graph ID.

Data structure	Type	Values	Comment
aipu_task_status_t	enum	AIPU_TASK_STATUS_NO_STATUS(0x0)	-
		AIPU_TASK_STATUS_RUNNING(0x1)	-
		AIPU_TASK_STATUS_DONE(0x2)	-
		AIPU_TASK_STATUS_EXCEPTION(0x3)	-

Data structure	Type	Values	Comment
aipu_status_t	enum	AIPU_STATUS_SUCCESS	-
		AIPU_STATUS_ERROR	-
		AIPU_STATUS_ERROR_INVALID_ARGS	-
		AIPU_STATUS_ERROR_GRAPH_NOT_EXIST	-
		AIPU_STATUS_ERROR_GRAPH_INIT_FAIL	-

		AIPU_STATUS_ERROR_INVALID_BIN	-
		AIPU_STATUS_ERROR_INVALID_BIN_HEAD_SIZE	-
		AIPU_STATUS_ERROR_INVALID_BIN_MAGIC	-
		AIPU_STATUS_ERROR_PARSE_BSS_STATIC_FAIL	-
		AIPU_STATUS_ERROR_PARSE_BSS_REUSE_FAIL	-
		AIPU_STATUS_ERROR_BUF_ALLOC_FAIL	-
		AIPU_STATUS_ERROR_INVALID_BUFFER	-
		AIPU_STATUS_ERROR_INVALID_GRAPH_VERSION	-
		AIPU_STATUS_ERROR_INIT_CONTEXT_FAIL	-
		AIPU_STATUS_ERROR_DEINIT_CONTEXT_FAIL	-
		AIPU_STATUS_ERROR_CONTEXT_NOT_INIT	-
		AIPU_STATUS_ERROR_TENSOR_NOT_ALLOC	-
		AIPU_STATUS_ERROR_TENSOR_NOT_EXIST	-
		AIPU_STATUS_ERROR_RUN_BUSY	-
		AIPU_STATUS_ERROR_RUN_EXCEPTION	-
		AIPU_STATUS_ERROR_RUN_SUCCESS_BUT_WARNING	-
		AIPU_STATUS_ERROR_GRAPH_IS_RUNNING	-
		AIPU_STATUS_ERROR_GRAPH_ID_NOT_RUN	-
		AIPU_STATUS_ERROR_GRAPH_NOT_MATCH_HARDWARE	-
		AIPU_STATUS_ERROR_PARSE_BSS_LINKED_FAIL	-
		AIPU_STATUS_ERROR_INVALID_IO_HANDLE	
		AIPU_STATUS_MAX	-

User application interfaces

Function declaration	void aipu_config_address(unsigned long ctrl_reg_base_addr, unsigned long memory_addr_offset);	
Parameter	ctrl_reg_base_addr	External control register base address.
	memory_addr_offset	Memory offset between the CPU and NPU.
Return value	void	
Comments	This API is used to set address information between the CPU and NPU. It is not thread safe.	

Function declaration	aipu_status_t aipu_init_ctx(void);
Parameter	void
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_INIT_CONTEXT_FAIL
Comments	This API is used to initialize the NPU context. It is not thread safe.

Function declaration	aipu_status_t aipu_deinit_ctx(void);
Parameter	void

Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_DEINIT_CONTEXT_FAIL
Comments	This API is used to destroy the NPU context. It is not thread safe.

Function declaration	aipu_status_t aipu_load_graph(void *graph, aipu_graph_desc_t *graph_desc);	
Parameter	graph	Graph binary buffer pointer.
	graph_desc	Graph descriptor.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_CONTEXT_NOT_INIT AIPU_STATUS_ERROR_BUF_ALLOC_FAIL AIPU_STATUS_ERROR_GRAPH_INIT_FAIL AIPU_STATUS_ERROR_INVALID_ARGS AIPU_STATUS_ERROR_INVALID_BIN_MAGIC AIPU_STATUS_ERROR_INVALID_BIN_HEAD_SIZE AIPU_STATUS_ERROR_INVALID_GRAPH_VERSION AIPU_STATUS_ERROR_GRAPH_NOT_MATCH_HARDWARE AIPU_STATUS_ERROR_PARSE_BSS_STATIC_FAIL AIPU_STATUS_ERROR_PARSE_BSS_REUSE_FAIL AIPU_STATUS_ERROR_PARSE_BSS_LINKED_FAIL	
Comments	This API is used to load a graph binary for the driver to parse and allocate static buffers. It is thread safe.	

Function declaration	aipu_status_t aipu_unload_graph(int graph_id);	
Parameter	graph_id	Graph ID.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_CONTEXT_NOT_INIT AIPU_STATUS_ERROR_GRAPH_NOT_EXIST AIPU_STATUS_ERROR_INVALID_ARGS AIPU_STATUS_ERROR_GRAPH_IS_RUNNING	
Comments	This API is used to unload a loaded graph. It is thread safe.	

Function declaration	aipu_status_t aipu_alloc_tensor_buffer(int graph_id, int io_buff_num);	
Parameter	graph_id	Graph ID.
	io_buff_num	Copies of tensor buffers.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_CONTEXT_NOT_INIT AIPU_STATUS_ERROR_GRAPH_NOT_EXIST AIPU_STATUS_ERROR_INVALID_ARGS AIPU_STATUS_ERROR_INVALID_BUFFER AIPU_STATUS_ERRO_BUF_ALLOC_FAIL	
Comments	This API is used to allocate buffers for all input and output tensors of a graph. The io_buff_num must be >=1, which indicates that how many copies of tensor buffers are allocated by runtime. After the API returns successfully, the valid tensor buffer handle is >=0 and < io_buff_num. If the io_buff_num is set to 1, the pipeline feature will be disabled. It is thread safe.	

Function declaration	aipu_status_t aipu_get_tensor_desc(int graph_id, int io_buff_handle, aipu_tensor_io_type_t io_type, aipu_tensor_desc_t* tdesc);	
Parameter	graph_id	Graph ID.
	io_buff_handle	The valid tensor buffer handle which is >=0 and < io_buff_num.
	io_type	Tensor IO type.
	tdesc	Tensor descriptor.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_CONTEXT_NOT_INIT AIPU_STATUS_ERROR_GRAPH_NOT_EXIST AIPU_STATUS_ERROR_TENSOR_NOT_ALLOC AIPU_STATUS_ERROR_INVALID_IO_HANDLE AIPU_STATUS_ERROR_TENSOR_INVALID_ARGS	
Comments	This API is used to get input/output/dump tensor descriptor of a graph. Applications load inputs and get outputs through these descriptors. It is thread safe.	

Function declaration	aipu_status_t aipu_start(int graph_id , int io_buff_handle);	
Parameter	graph_id	Graph ID.
	io_buff_handle	Valid buffer handle which has loaded inputs for aipu_v1/v2. It is useless for aipu_v3, and recommended to use value 0.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_CONTEXT_NOT_INIT AIPU_STATUS_ERROR_GRAPH_NOT_EXIST AIPU_STATUS_ERROR_TENSOR_NOT_ALLOC AIPU_STATUS_ERROR_INVALID_IO_HANDLE AIPU_STATUS_ERROR_RUN_BUSY AIPU_STATUS_ERROR_RUN_EXCEPTION AIPU_STATUS_ERROR_RUN_SUCCESS_BUT_WARNING	
Comments	• Before calling this API, putting aipu_alloc_tensor_buffer() and tensor data into tdes should be done. • aipu_start() and aipu_get_status() should be one-to-one, otherwise, the status will be covered by the next running graph. • If this API returns the exception state, the application can call the aipu_get_status() API to clear the exception status. • It is thread safe.	

Function declaration	aipu_status_t aipu_get_status(int graph_id, aipu_task_status_t *status);	
Parameter	graph_id	Graph ID.
	status	NPU task status.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_CONTEXT_NOT_INIT AIPU_STATUS_ERROR_GRAPH_NOT_EXIST AIPU_STATUS_ERROR_GRAPH_ID_NOT_RUN AIPU_STATUS_ERROR_RUN_EXCEPTION	
Comments	• This API is non-blocking, so the application needs to poll this API to get status.	

	• When this API finds that the NPU runs into exception state, the API will clear the exception state, and then return the exception state value. • It is thread safe.

Function declaration	aipu_status_t aipu_free_tensor_buffer(int graph_id);	
Parameter	graph_id	Graph ID.
Return value	AIPU_STATUS_SUCCESS AIPU_STATUS_ERROR_CONTEXT_NOT_INIT AIPU_STATUS_ERROR_GRAPH_NOT_EXIST	
Comments	This API is used to free the tensor buffer. It is thread safe.	

1.5.3 RTOS platform programming model

A programming model in bare metal is also suitable for RTOS. For the usage of the interfaces, see *5.4.2 Bare-metal platform programming model*. In addition, RTOS is an operating system which supports the multi-task feature that is not supported in bare metal. Therefore, a multi-task programming model is only applicable to RTOS.

Code 28 shows a multi-task programming model. The main task is responsible for the NPU context initialization and de-initialization, and each subtask loads the graph and runs the inference job. Before the subtask loads the graph, the main task must complete the context initialization first.

```
#define AIPU_CTRL_REG_BASE_ADDR (0x64000000u)
#define MEMORY_ADDR_OFFSET  (0x80000000u)
/* main task */
void main()
{
    /* config address of cpu and aipu */
    aipu_config_address(AIPU_CTRL_REG_BASE_ADDR, MEMORY_ADDR_OFFSET);
    aipu_status_t ret = AIPU_STATUS_SUCCESS;
    /* initialize context */
    ret = aipu_init_ctx();

    /* waiting all subtasks complete and exit */
    /* ... */

    /* de-init context */
    ret = aipu_deinit_ctx();
}

/* task 1 */
void task1()
{
    /* load graph */
    /* alloc tensor buffer */
    /* get tensor descriptor entity from driver */
    /* load inputs */
    /* start aipu and get status*/
    /* ... */
    /* free all tensor buffer */
    /* unload graph */
}

/* task 2 */
void task2()
{
    /* load graph */
```

```
    /* alloc tensor buffer */
    /* get tensor descriptor entity from driver */
    /* load inputs */
    /* start aipu and get status*/
    /* ... */
    /* free all tensor buffer */
    /* unload graph */
}

/* task n */
/* ... */
```

Code 28. NPU Driver Library Use Case – for RTOS (Part 1)

Chapter 2
NPU runtime

This chapter describes the Zhouyi NPU runtime.

It contains the following sections:

2.1 About the NPU runtime

The Zhouyi NPU runtime is a program that provides a runtime system for the *Neural Network* (NN). It sets up the running environment, with the handle of the NPU and maybe some other hardware, along with their drivers. In the NPU runtime, it has a hardware abstraction for the NPU and other hardware, to treat different hardware with the same interfaces as the NPU.

The NPU runtime also provides a set of application interfaces, with which engineers can build their neural network applications.

The Zhouyi Compass NPU runtime is developed based on the Arm NN interference engine and Android NNAPI.

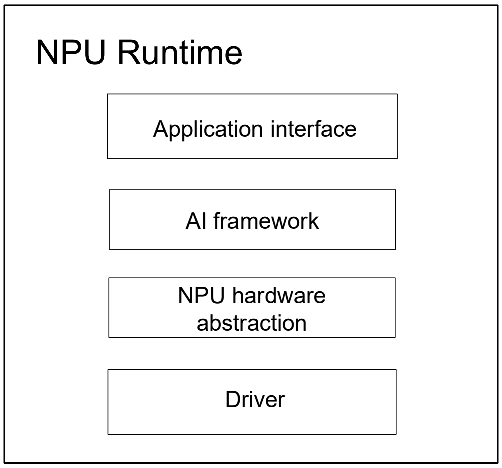

Figure 2-1 NPU runtime architecture

2.2 Arm NN runtime

The Arm NN runtime for Zhouyi NPUs contains the original Arm NN framework and the Zhouyi backend.

Arm NN is an inference engine for CPUs, GPUs and NPUs. It bridges the gap between existing NN frameworks and the underlying IPs. It enables efficient translation of existing NN frameworks such as TFLite and ONNX, allowing them to run efficiently, without modification, across Arm Cortex-A CPUs, Arm Mali GPUs, and third-party NPUs. Based on the Arm NN framework, the Zhouyi backend is added to support the inferences on Zhouyi NPUs. The Zhouyi backend translates and schedules Arm NN inference tasks by calling the APIs provided by the NPU build tool and driver. Currently, the Arm NN Zhouyi backend supports AIPU V2 and V3 NPUs.

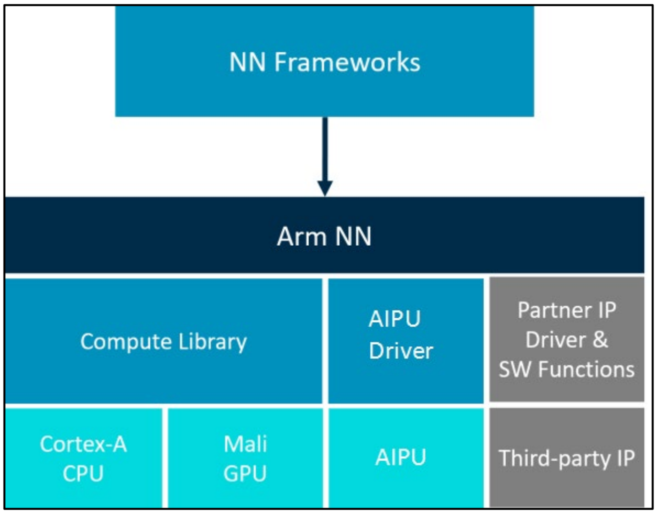

Figure 2-2 Arm NN architecture

2.2.1 Building Arm NN with the Zhouyi backend

The Arm NN version used is 23.08 with no modification. You can download the Arm NN source (and dependencies of it) from GitHub and get the Zhouyi NPU backend from the release package.

The Zhouyi NPU backend supports both x86-linux and arm64-linux environments. The x86-linux Arm NN is only suggested to be used in your software bring-up stage and it depends on the Zhouyi NPU simulators.

Generally, Arm NN with the Zhouyi backend depends on the binary modules as shown in the following table.

Table 2-1 Dependent software modules

Module	Description	Source
`libaipubuildtool.so`	Zhouyi build tool lib	Arm China Compass Software release package
`libaipuopplugin.so`	Zhouyi operator plugin lib	
`libaipu_xlib.so`	Zhouyi NPU operator layer lib	
`aiffcllib.a` & `tpccllib.a`	Zhouyi AIPU V3 NPU operator layer lib	
`aipuoas`, `aipuold, libAIPUOGB.so, libaiputoolchain.so`	Zhouyi compiler toolchains (for AIPUv3 only)	
Zhouyi NPU simulator binaries and libraries (for x86-linux simulation only)	-	

Module	Description	Source
`aipu.ko`	Zhouyi NPU KMD driver (for arm64-linux only)	Built from the released Compass driver source code
`libaipudrv.so`	Zhouyi NPU UMD driver	
Dependencies of Arm NN (such as TensorFlow, and FlatBuffer)	-	Downloaded as described in the Arm NN documentation

To run an application with Arm NN, you also need to build an Arm NN TFLite parser and the test applications of it. You can download them from the Arm NN GitHub repository. Note that currently only quantized TFLite models are accepted.

For detailed instructions on how to integrate the Zhouyi NPU backend into Arm NN and test the applications, see the `README.md` file in the Zhouyi NPU backend source directory. To simplify the integration flow, Arm China also provides an example `build.sh` script in the release package. You can find how to configure and use the build script in the readme file and the script itself.

2.2.2 Running the sample application

You can find three mobilenet-v2 sample applications (one of them is a multi-batch case) in `AI610-SDK-1014/armnn-runtime/sample` for x86-linux environments.

Follow the steps in `AI610-SDK-1014/armnn-runtime/README` to run the sample applications on different target platforms.

If you want to use the auto-tiling feature provided by the Zhouyi build tool, enable it by configuring the environment variable TILING_KEY to be one of the following tiling methods before executing your Arm NN application:

```
$ export TILING_KEY=fps
```

2.3 Android neural networks runtime

The *Neural Network* (NN) HAL defines an abstraction of various devices, such as *Graphics Processing Units* (GPUs) and *Digital Signal Processors* (DSPs), that are in a product (for example, a phone or a tablet). The drivers for these devices must conform to the NN HAL. As a vendor device, the Zhouyi NPU complies with the API of Android NN HAL from HIDL 1.0~1.3, and AIDL from Android 12.

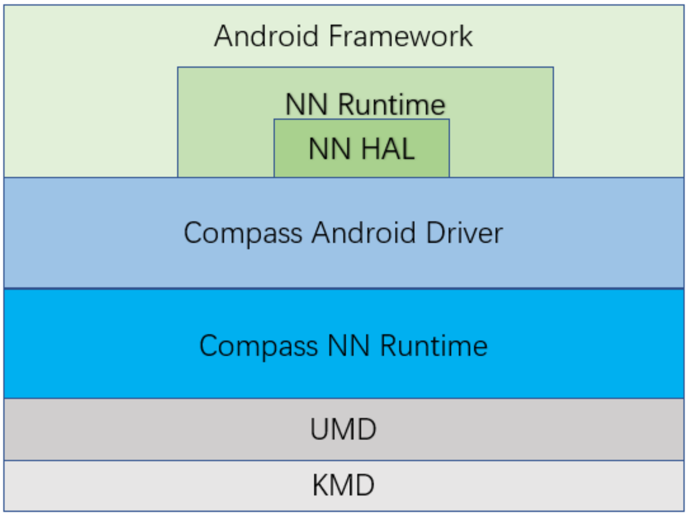

Figure 2-3 Android NN Runtime architecture

2.3.1 Using the Android Neural Networks driver

This section describes how to use the NPU driver for the Android Neural Networks API to implement the android.hardware.neuralnetworks HALs.

Before you begin

Check the AOSP version in P/Q/R/S/T.

Files provided

You can find the following file structure in the software package:

```
Compass_Android_Driver/nnapi/
|-- hidl
|-- aidl (from Android12)
|-- LICENSE.TXT
|-- NnapiSupport.txt (support operator list)
|-- README.md
|-- sepolicy_sample
`-- test (test apk)

Compass_NN_Runtime/
|   |-- aipulib
|   |-- Android.mk
|   |-- build_libs_ndk.sh
|   |-- CMakeLists.txt
```

```
|   |-- config.mk
|   |-- include
|   |-- LICENSE.TXT
|   |-- OperatorsSupport.txt
|   |-- README.md
|   |-- samples
|   `-- src
Dependencies/
Simulator/ (executable files only for V2)

Prebuilt aipulib/
|   |-- libaipu_buildingtool.so
|   |-- v2
|   |   |-- libaipu_driver.so
|   |   `-- libaipu_layerlib.so
|   `-- v3
|       |-- libaipu_driver.so
|       |-- libaipu_layerlib.so
|       |-- libaiputoolchain_core.so
|       `-- libaiputoolchain.so
```

Procedure

1. Prepare AOSP source tree `<ANDROID_ROOT>`.
2. Create the driver directory `<AOSP_ROOT>/hardware/armchina/neuralnetworks/`, and then copy `<RELEASE_PACKAGE>/Compass_Android_Driver` into it. If you do not attempt to build AIDL, you can delete `Compass_Android_Driver/nnapi/aidl`.
3. Copy `<RELEASE_PACKAGE>/Compass_NN_Runtime` into `<AOSP_ROOT>/hardware/armchina/neuralnetworks/`.
4. Update the Android build environment to add the Zhouyi driver.

   Add `PRODUCT_PACKAGES` to `vendor.mk` in `<ANDROID_ROOT>/device/<manufacturer>/<product>`, for example, if you try to run on the x86_64 emulator, it is `<ANDROID_ROOT>/device/generic/goldfish/vendor.mk`.

   For HAL HIDL 1.3:

   ```
   PRODUCT_PACKAGES += android.hardware.neuralnetworks@1.3-service-zhouyi \
          libcompass_nn_runtime \
          <prebuilt shared libraries>
   ```

   For AIDL:

   ```
   PRODUCT_PACKAGES += android.hardware.neuralnetworks-service-zhouyi \
                       libcompass_nn_runtime \
                       <prebuilt shared libraries>
   ```

5. Set the configuration file under `<RELEASE_PACKAGE>/Compass_NN_Runtime/config.mk`.

- **AIPU_VERSION**: If you set it to v2, the default target will be X1_1204.

  If you set it to v3, the default target is X2_1204. If you want to use X2_1204 in a 3-core configuration, the target name is X2_1204MP3.

  When you try to specify other targets such as Z3_1204, you can use the following:

  ```
  $ adb -s <emulator_name> shell setprop vendor.nn.zhouyi.simulator_target
  Z3_1204
  ```

  For hardware users, AIPU_VERSION must be consistent with the actual hardware, and cannot be modified by `setprop`.
- **USE_AIPULIB_LOCAL**: Set to 1.
- **BUILD_STANDALONE_LIB**: If you try to build `libcompass_nn_runtime.so` separately such as for delegate, set it to 1, otherwise set it to 0.

6. Build AOSP. For details, visit https://source.android.com/setup/build/building.
7. To confirm that the Zhouyi driver has been built, check the driver service at `<ANDROID_ROOT>/out/target/product/<product>/vendor/bin/hw/android.hardware.neuralnetworks@1.3-service-zhouyi` or `android.hardware.neuralnetworks-service-zhouyi` for AIDL, and check the shared library in `<ANDROID_ROOT>/out/target/product/<product>/vendor/lib64/libcompass_nn_runtime.so` and shared libraries in the prebuilt list.

Note

For Android emulator + AIPU simulator users, the AIPUv2 simulator is an executable file, while the AIPUv3 simulator is a shared library.

When the lunched `TARGET_ARCH` is `x86_64`, the built files will run on the AIPU simulator by default. If it is `arm64`, they will run on hardware.

For HAL users, it is best to restart (stop|start) the service if you want to enable the new property:

```
- hidl: neuralnetworks_hal_service_zhouyi
- aidl: neuralnetworks_hal_service_aidl_zhouyi
```

2.3.2 Running the driver on the emulator

1. Launch the emulator with writable, then root, and remount with `adb`.
2. Push relative files into the Android emulator.
   - For the AIPUv2 architecture:

     ```
     $ adb -s <emulator_name> shell mkdir -p /vendor/etc/npu/
     $ adb -s <emulator_name> push <RELEASE_PACKAGE>/simulator /vendor/etc/npu/
     $ adb -s <emulator_name> shell chmod 777 -R /vendor/etc/npu/simulator
     ```
   - For the AIPUv3 architecture:

     ```
     $ adb -s <emulator_name> shell mkdir -p /vendor/etc/npu/operators/
     $ adb -s <emulator_name> push <RELEASE_PACKAGE>/dependencies/*.a
     /vendor/etc/npu/operators/
     ```

   Where, `<RELEASE_PACKAGE>` is the package directory. `<emulator_name>` is the current emulator.
3. The service will check the above files at startup. If this is the first time you prepare them, you can reboot the emulator or pack these files into the Android image directly.

2.3.3 Using the profiler feature on Android

This section describes how to use the profiler feature on Android. Before you proceed, ensure that you have completed the instructions in *6.3.1 Using the Android Neural Networks driver*.

Note

You cannot enable/disable the profiler when you are running it, so you need to enable/disable the profiler before you start up the driver service. The profiler is disabled by default.

Perform the following steps to use the profiler feature:

1. Set properties before running the model.

   ```
   $ adb -s <emulator_name> setprop vendor.nn.zhouyi.profile <on/off>
   ```

2. Run the model, such as `pineapple_test.apk` as described in *2.3.5 Testing*.
3. Fetch profiler files on the simulator (the simulator shares one `aiff_dumps` directory).

   ```
   $ adb -s <emulator_name> pull /data/user/nnapi/aiff_dumps
   ```

   ```
   $ adb -s <eumulator_name> pull /data/user/nnapi/graph_1/graph.json
   ```

4. Fetch profiler files on the hardware.

   ```
   $ adb -s <device_name> pull /data/user/nnapi/graph_<n>/graph.json
   ```

   ```
   $ adb -s <device_name> pull /data/user/nnapi/graph_<n>/temp.profile
   ```

For information on how to analyze the profile data, see *6.2 Using the aipu_profiler* in the *Arm China Zhouyi Compass NN Compiler User Guide*.

2.3.4 Using the tiling feature on Android

1. Run the following command to enable/disable the tiling feature.

   ```
   $ adb -s <emulator_name> shell setprop vendor.nn.zhouyi.tiling <fps/disable>
   ```

2.3.5 Testing

1. Install the application.

   ```
   $adb -s <emulator_name> install -t <RELEASE_PACKAGE>/test/pineapple_test.apk
   ```

   ```
   $adb -s <emulator_name> shell am start
   com.example.tflite_imageclassifier/.MainActivity
   ```

2. Execute.
3. Check whether Android NN uses Zhouyi as the backend.

   ```
   $ adb -s <emulator_name> shell setprop vendor.nn.zhouyi.vlog on
   ```

4. Click **Detect** to begin inferring.
   A moment later, you can see the result on the application.
5. After inference, run `logcat` as follows:

   ```
   $ adb -s <emulator_name> logcat
   ```

You should find ‘ZhouyiDriver’ and ‘EXECUTION WAS SUCCESSFUL’ in the log.

2.4 TensorFlow Lite delegate runtime

TensorFlow Lite delegates enable hardware acceleration of TensorFlow Lite models by leveraging on-device accelerators such as the NPU, GPU and *Digital Signal Processor* (DSP).

By default, TensorFlow Lite utilizes CPU kernels that are optimized for the Arm Neon instruction set. However, the CPU is a multi-purpose processor that is not necessarily optimized for the heavy arithmetic typically found in Machine Learning models (for example, the matrix math involved in convolution and dense layers). As most modern mobile phones contain chips that are better at handling these heavy operations, utilizing them for neural network operations provides huge benefits in terms of latency and power efficiency. Each of these accelerators have associated APIs that enable custom computations. TensorFlow Lite's Delegate API acts as a bridge between the TFLite runtime and these lower-level APIs.

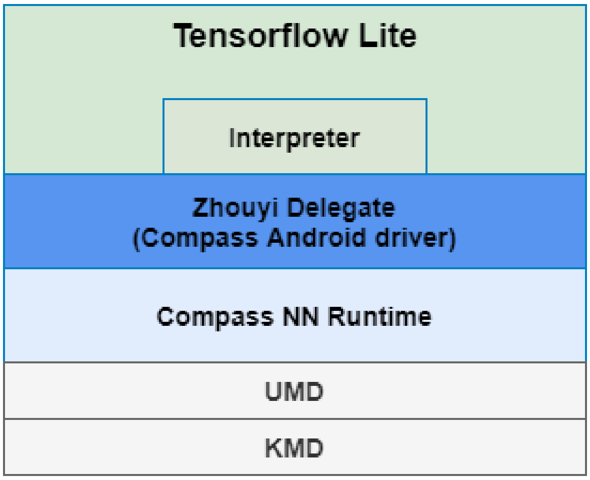

Figure 2-4 TensorFlow Lite Runtime architecture

This section describes how to use the TensorFlow Lite Zhouyi Delegate in your application using Java and/or C APIs.

The delegate leverages the Zhouyi library to execute quantized kernels on the NPU.

Currently the following devices are tested:

- Android x86_64 emulator: Runs on the NPU simulator.
- Juno: Runs on the NPU hardware.

Currently the following Zhouyi architecture is tested:

- v3

The Zhouyi delegate supports all models that conform to Arm China quantization specification, including symmetric quantization models and asymmetric quantization models.

2.4.1 How to build

- Set up the environment.
    a. Get the TensorFlow code from GitHub and check out the branch to r2.12.
    b. Ensure that `tensorflow-lite.aar` can be compiled.
    c. Get the Zhouyi delegate code from the release package. You can find the following file structure in the software package:

```
Compass_Android_Driver/zhouyi
|-- BUILD
|-- builders
|-- compass_nn_runtime
```

```
|-- docs
|-- java
|-- utils.cc
|-- utils.h
|-- version_script.lds
|-- zhouyi_delegate.cc
|-- zhouyi_delegate.h
|-- zhouyi_delegate_kernel.cc
|-- zhouyi_delegate_kernel.h
|-- zhouyi_implementation.cc
`-- zhouyi_implementation.h
```

d. Put the `zhouyi` folder in the following path:

`tensorflow/lite/delegates/.`

- Build the AAR package.

  After Bazel is properly configured, you can build the Zhouyi delegate AAR from the root checkout directory as follows:

```
bazel build -c opt --config=<arch>
tensorflow/lite/delegates/zhouyi/java:tensorflow-lite-zhouyi
```

  This will generate an AAR file in `bazel-bin/tensorflow/lite/delegates/zhouyi/java/`.

- Build the delegate.

```
bazel build -c opt --config=<arch>
tensorflow/lite/delegates/zhouyi:zhouyi_delegate
```

  This will generate a `libzhouyi_delegate.so` file in `bazel-bin/tensorflow/lite/delegates/zhouyi`.

- Build tests.

```
bazel --bazelrc=/dev/null build -c opt --config=<arch>
tensorflow/lite/delegates/zhouyi/builders/tests:all
```

  This will generate the *zhouyi_xxx* test file in `bazel-bin/tensorflow/lite/delegates/zhouyi/builders/tests`.

- Implement the architecture support:
  - android_x86_64
  - android_arm64

2.4.2 Zhouyi delegate Java API

```
public class ZhouyiDelegate implements Delegate, Closeable {
  /**
  * Zhouyi Delegate DMA buffer
  * Use this object to access zhouyi's dma buffer.
  */
  public static final class DMABuffer implements Closeable {
    /**
    * init DMABuffer
    */
    public DMABuffer(long size)

    /**
```

```
    * get buffByteBuffer
    */
    public ByteBuffer bytebuffer()

    /**
    * get buffer size
    */
    public long size

    /**
    * Bind buffer handle to tensor
    */
    public long bind(long delegateHandle, int tensor_index)

    /**
     * Explicit call to release resources.
     */
    @Override
    public void close()
  }

  /** Delegate options. */
  public static final class Options {
    /**
     * Enable or disable the Zhouyi runtime dump file.
     * default settings: false.
     */
    public Options setEnableDump(boolean enable)

    /**
     * Enable or disable the Zhouyi runtime profile.
     * It generates performance files
     * default settings: false.
     */
    public Options setEnableProfile(boolean enable)

    /**
     * Set the Zhouyi runtime tiling method, it support "fps" "footprint".
     * Titing is a model segmentation function.
     * default settings: none.
     */
    public Options setTiling(String tilingMethod)

    /**
     * Configure the location to be used to store model compilation cache entries.
     * If not set, the default is "/data/tata/{app package name}"
     */
    public Options setCacheDir(String cacheDir)

     /**
     * This corresponds to the log level in the Zhouyi Runtime.
     * 0(defalut): LOG_ERROR
     * 1: LOG_WARN
     * 2: LOG_INFO
     * 3: LOG_DEBUG
     * 4: LOG_VERBOSE
     */
    public Options setLogLevel(int level)

    /**
     * Returns whether the Zhouyi runtime "dump file" function is enabled.
```

```
     */
    public Boolean getEnableDump()
  }

  /*
   * Creates a new ZhouyiDelegate object without param.
   * Throws UnsupportedOperationException if Zhouyi AIPU delegation is not available on this device.
   */
  public ZhouyiDelegate() throws UnsupportedOperationException

  /*
   * Creates a new ZhouyiDelegate object given the current 'options'.
   * Throws UnsupportedOperationException if Zhouyi AIPU delegation is not available
   * on this device.
   */
  public ZhouyiDelegate(Options setOptions) throws UnsupportedOperationException

  /**
   * Frees TFLite resources in C runtime.
   * User is expected to call this method explicitly.
   */
  @Override
  public void close();
}
```

Example usage

1. Add the AAR file to the application.
2. Add `tensorflow-lite-zhouyi-xxx.aar` in the release package to the `app/libs/` directory of the application.
3. Edit `app/build.gradle` to use the Zhouyi delegate AAR.

```
dependencies {
  ...
  implementation 'org.tensorflow:tensorflow-lite:xx.xx.xx'
  implementation files('libs/tensorflow-lite-zhouyi-xxx.aar')
}
```

Note

The AAR file can also be added using the Android Studio tool.

4. Add Zhouyi libraries to your Android app. See *2.4.5 Adding shared libraries to an app*.

Note

The Zhouyi shared library must be used with the AAR.

5. Create a delegate and initialize a TensorFlow Lite Interpreter.

```
import org.tensorflow.lite.ZhouyiDelegate;
private ZhouyiDelegate zhouyidelegate = null;
private Interpreter tfliteInterpreter;
private Interpreter.Options tfliteOptions = new Interpreter.Options();
```

```
// Create the Delegate instance.
try {
  ZhouyiDelegate.Options zhouyiDelegateOptions = new
ZhouyiDelegate.Options();
  zhouyiDelegateOptions.setLogLevel(4);
  zhouyiDelegateOptions.setEnableProfile(false);
  zhouyiDelegateOptions.setEnableDump(false);
  zhouyiDelegateOptions.setTiling("fps");
if(zhouyidelegate == null) {
    zhouyidelegate = new ZhouyiDelegate(zhouyiDelegateOptions);
  }
  tfliteOptions.addDelegate(zhouyidelegate);
} catch (UnsupportedOperationException e) {
  // Zhouyi delegate is not supported on this device.
}

tfliteInterpreter = new Interpreter(tfliteModel, tfliteOptions);

//...
//work
//...


// Dispose after finished with inference.
tfliteInterpreter.close();
if (zhouyiDelegate != null) {
  zhouyiDelegate.close();
}
```

2.4.3 Zhouyi delegate C API

```
// Use TfLiteZhouyiDelegateOptionsDefault() for Default options.
struct TFL_CAPI_EXPORT TfLiteZhouyiDelegateOptions {
// This determines whether Zhouyi runtime dumps intermediate files.
  // false(defalut)
  bool enableDump;

  // This corresponds to the opening of the Zhouyi runtime profile function.
  // false(defalut)
  bool enableProfile;

  // This corresponds to the log level in the Zhouyi Runtime.
  // 0(defalut): LOG_ERROR
  // 1: LOG_WARN
  // 2: LOG_INFO
  // 3: LOG_DEBUG
  // 4: LOG_VERBOSE
  int logLevel;

  // This corresponds to the opening of the Zhouyi runtime tiling function.
  // Only support "footprint", "fps".
  char tilingMethod[512];

  // Current,the compilation process needs to generate intermediate files,
  // it is necessary to specify a directory to store intermediate files.
  // This directory usually is "/data/data/{package name}".
  char cacheDir[512];

  // This sets the maximum number of Zhouyi graphs created with
  // zhouyi_nn_init. Each graph corresponds to one delegated node subset in
the
```

```
  // TFLite model.
  int max_delegated_partitions;
  // This sets the minimum number of nodes per graph created with
  // zhouyi_nn_init. Defaults to 2.
  int min_nodes_per_partition;
};

// Returns TfLiteZhouyiDelegateOptions populated with default values.
TFL_CAPI_EXPORT TfLiteZhouyiDelegateOptions
TfLiteZhouyiDelegateOptionsDefault();

// Same as above method but doesn't accept the path params.
// Return a delegate that uses Zhouyi SDK for ops execution.
// Must outlive the interpreter.
TfLiteDelegate* TFL_CAPI_EXPORT
TfLiteZhouyiDelegateCreate(const TfLiteZhouyiDelegateOptions* options);

// Do any needed cleanup and delete 'delegate'.
void TFL_CAPI_EXPORT TfLiteZhouyiDelegateDelete(TfLiteDelegate* delegate);

// Assumes the environment setup is already done. Only initialize Zhouyi.
void TFL_CAPI_EXPORT TfLiteZhouyiInit();

// Clean up and switch off the AIPU.
// This should be called after all processing is done and delegate is deleted.
void TFL_CAPI_EXPORT TfLiteZhouyiTearDown();

//Allocate zhouyi dma buffer.
int TFL_CAPI_EXPORT
TfLiteZhouyiAllocateDMABuffer(int size);

//Get buffer addr.
void* TFL_CAPI_EXPORT
TfLiteZhouyiGetAddr(int buffer_handle);

//Free zhouyi dma buffer by handle.
void TFL_CAPI_EXPORT
TfLiteZhouyiFreeDMABuffer(int buffer_handle);

//Free zhouyi dma buffer by pointer.
void TFL_CAPI_EXPORT
TfLiteZhouyiFreeDMABufferByAddr(void* buffer);

// Bind DMA buffer to I/O tensor.
// Note that this function must be called after Interpreter::ModifyGraphWithDelegate().
TfLiteStatus TFL_CAPI_EXPORT
TfLiteZhouyiBindBufferToTensor(
    TfLiteDelegate* delegate, int buffer_handle, int tensor_index);

// Bind DMA buffer to I/O tensor by tensor pointer.
// Note that this function must be called after Interpreter::ModifyGraphWithDelegate().
TfLiteStatus TFL_CAPI_EXPORT
TfLiteZhouyiBindBufferToTensorP(
    TfLiteDelegate* delegate, int buffer_handle, TfLiteTensor* tensor);

```

Example usage

1. Add the AAR file to the application.

2. Add `tensorflow-lite-zhouyi-xxx.aar` in the release package to the `app/libs/` directory of the application.
3. Edit `app/build.gradle` to use the Zhouyi delegate AAR.

```
dependencies {
  ...
  implementation 'org.tensorflow:tensorflow-lite:xx.xx.xx'
  implementation files('libs/tensorflow-lite-zhouyi-xxx.aar')
}
```

Note

The AAR file can also be added using the Android Studio tool.

4. Add Zhouyi libraries to your Android app. See *2.4.5 Adding shared libraries to an app*.

Note

The Zhouyi shared library must be used with the AAR.

5. Include the C header.

   The header file `zhouyi_delegate.h` can be extracted from the Zhouyi delegate AAR.

6. Create a delegate and initialize a TensorFlow Lite Interpreter.

   In your code, ensure that the native Zhouyi library is loaded. This can be done by calling in your Activity or Java entry-point.

```
System.loadLibrary("tensorflowlite_zhouyi_jni");
```

Example of creating a delegate:

```
#include "tensorflow/lite/delegates/zhouyi/zhouyi_delegate.h"

// If files are packaged with native lib in android App then it
// will typically be equivalent to the path provided by
// "getContext().getApplicationInfo().nativeLibraryDir"
const char[] library_directory_path = "/data/data/{package name}";
TfLiteZhouyiInit();

::tflite::TfLiteZhouyiDelegateOptions options = {
    .cacheDir = library_directory_path,
  };
// 'delegate_ptr' Need to outlive the interpreter. For example,
// If use case will need to resize input or anything that can trigger
// re-applying delegates then 'delegate_ptr' need to outlive the interpreter.
auto* delegate_ptr = ::tflite::TfLiteZhouyiDelegateCreate(&options);
Interpreter::TfLiteDelegatePtr delegate(delegate_ptr,
  [](TfLiteDelegate* delegate) {
    ::tflite::TfLiteZhouyiDelegateDelete(delegate);
  });
interpreter->ModifyGraphWithDelegate(delegate.get());
// After usage of delegate.
TfLiteZhouyiTearDown();  // Needed once at end of app/Zhouyi usage.
```

2.4.4 Supported ops

Refer to the `Supported Ops.md` file in the code package.

2.4.5 Adding shared libraries to an app

Generally, the Zhouyi delegate depends on the binary modules as shown in the following table.

Table 2-2 Delegate dependencies

Module	Description	Source
`libaipu_buildingtool.so`	Zhouyi build tool lib	Arm China Compass Software release package
`libaiputoolchain.so`	Zhouyi operator plugin lib	
`libaiputoolchain_core.so`	Zhouyi NPU operator layer lib	
`libaipugenerate_afile.so`	Zhouyi AIPU V3 NPU operator layer lib	
`libaipu_layerlib.so`		
`libaipu_simulator_x2.so`	For Android x86_64 emulator only	
`libcompass_nn_runtime.so`	Compass NN runtime lib	Arm China Compass Software release package or built from the released Compass NN runtime source code
`libaipu_driver.so`	Zhouyi NPU UMD driver	Arm China Compass Software release package or built from the released Compass driver source code
`aipu.ko`	Zhouyi NPU KMD driver (for arm64 only)	Built from the released Compass driver source code

The release package provides libraries for the following two target architectures:

- x86_64
- arm64-v8a

To add shared libraries to an application

1. Find the file of the corresponding architecture, and add it to the `app/libs/{arch}` path.
2. Modify `build.gradle` of the application to add the following code:

```
sourceSets {
        main {
            jniLibs.srcDirs = ['libs']
        }
    }
```

2.4.6 Input/Output buffers

The Zhouyi delegate provides a set of C++/Java APIs to directly read from and write to the NPU hardware buffer, bypassing avoidable memory copies.

C++ API usage

The following techniques are only available when you are using Bazel or building TensorFlow Lite yourself.

- The function must be called after `Interpreter::ModifyGraphWithDelegate()`. In addition, the inference output is copied from NPU memory to CPU memory by default. You can turn this behavior off by calling `Interpreter::SetAllowBufferHandleOutput(true)` during initialization.

- If the input of the network is already loaded in the NPU memory, the input buffer handle can be directly bound to the input tensor.
- The output data of the network can also be used directly as the input of the next network.
- The buffer data is released when the subgraph is destroyed, and it also can be released by calling interface `TfLiteZhouyiFreeDMABuffer`.

C++ API example usage

```
#include "tensorflow/lite/delegates/zhouyi/zhouyi_delegate.h"
// ...
// Prepare NPU delegate.

// If files are packaged with native lib in android App then it
// will typically be equivalent to the path provided by
// "getContext().getApplicationInfo().nativeLibraryDir"
const char[] library_directory_path = "/data/local/tmp";
TfLiteZhouyiInit();

::tflite::TfLiteZhouyiDelegateOptions options = {
    .cacheDir = library_directory_path,
  };
auto* delegate_ptr = ::tflite::TfLiteZhouyiDelegateCreate(&options);
Interpreter::TfLiteDelegatePtr delegate(delegate_ptr,
  [](TfLiteDelegate* delegate) {
    ::tflite::TfLiteZhouyiDelegateDelete(delegate);
  });
interpreter->ModifyGraphWithDelegate(delegate.get());

interpreter->SetAllowBufferHandleOutput(true);  // disable default npu->cpu copy
int user_provided_input_buffer = TfLiteZhouyiAllocateDMABuffer(input_size);
int user_provided_output_buffer = TfLiteZhouyiAllocateDMABuffer(output_size);
if (!TfLiteZhouyiBindBufferToTensor(
        delegate, interpreter->inputs()[0], user_provided_input_buffer)) {
  return false;
}
if (!TfLiteZhouyiBindBufferToTensor(
        delegate, interpreter->outputs()[0], user_provided_output_buffer)) {
  return false;
}

// Run inference.
if (interpreter->Invoke() != kTfLiteOk) return false;
```

Java API usage

By default, the inference output is copied from NPU memory to CPU memory. You can turn this behavior off by calling `Interpreter.Options.setAllowBufferHandleOutput(true)` during initialization.

JAVA API example usage

```
// interpreter prepare
// delegate prepare

// ...

// input
ZhouyiDelegate.DMABuffer dmaBufferIn = new ZhouyiDelegate.DMABuffer(input_size);
```

```
ByteBuffer bytebuffer = dmaBufferIn.bytebuffer();
dmaBufferIn.bind(delegate.getNativeHandle(),
interpreter.getInputTensor(0).index());
// copy input data to bytebuffer


// output
ZhouyiDelegate.DMABuffer dmaBufferOut = new
ZhouyiDelegate.DMABuffer(output_size);
dmaBufferOut.bind(delegate.getNativeHandle(),
interpreter.getOutputTensor(0).index());
// ...

// the output data:
ByteBuffer output_bytebuffer = dmaBufferOut.bytebuffer();

// after invoke
dmaBufferIn.close();
dmaBufferOut.close();
```

2.4.7 Testing

- OP test
  1. Push shared libraries to device path `/vendor/lib64/`.
  2. Push *zhouyi_xxx_test* to device path `/data/local/tmp`.
  3. Run *zhouyi_xxx_test*.
     For the compilation instructions of *zhouyi_xxx_test*, see *2.4.2 Zhouyi delegate Java API*.
- Sample app test
  1. Get the `delegateJavaSample` code from the release package.
  2. Build the app and install it onto the device.
  3. Run the app. If you use the Juno device, install the driver first, that is, `aipu.ko`.

2.4.8 FAQs

The following information provides answers to general frequently asked questions.

How can I tell that the model is using Zhouyi when I enable the delegate?

Two log messages will be displayed when you enable the delegate:

- One is to indicate whether the delegate was created.
- The other is to indicate how many nodes are running using the delegate.

```
Created TensorFlow Lite delegate for Zhouyi.
Zhouyi delegate: X nodes delegated out of Y nodes.
```

Do all Ops in the model need to be supported to run the delegate?

No. The model will be partitioned into subgraphs based on the supported ops. Any unsupported ops will run on the CPU.

System permission: I have rooted my phone but still cannot run the delegate, what should I do?

Be sure to disable SELinux enforcing by running `adb shell setenforce 0`.