壹 普 智 慧

EEP-TPU Application Programming Interface (API) User Manual

eep-ug053 (v0.1.0)

2023-02-01

www.embedeep.com

Revision history:

Version	Date	Describe	Author
0.1.0	2023-02-01	Initial version	He

Contents

1. Introduction

This document is a description of the "libeeptpu_pub.so" library. You can use this document to learn about EEP-TPU operations.

The libeeptpu pub library must work with the public version of the bin file. When compiling a neural network through the EEP-TPU compiler， add the "--public_bin" parameter to generate a bin file in the format of ".pub.bin".

1.1 Compilation Environment

arm-32 bit

Cross-compile toolchain: arm-linux-gnueabihf-g++, gcc version 6.3.1 20170404 (Linaro GCC 6.3-2017.05). Download address of the cross-compilation toolchain:

http://releases.linaro.org/components/toolchain/binaries/6.3-2017.05/arm-linux-gnueabihf/gcc-linaro-6.3.1-2017.05-i686_arm-linux-gnueabihf.tar.xz

aarch-64 bit

Cross-compile toolchain: aarch64-linux-gnu-g++, gcc version 6.3.1 20170404 (Linaro GCC 6.3-2017.05). Download address of the cross-compilation toolchain:

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

X86 platform

Compiler: g++, gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04) (Ubuntu 18.04.6 LTS)

Note: The cross-compilation toolchain used by the program needs to be consistent with the above version, otherwise some compatibility issues may occur. The database file libeeptpu_pub.so v0.7.0 or later, which must be used with eeptpu_compiler v2.4.1 or later, and EEP-TPU hardware version v0.8.4.

2. API interface

2.1 Initialization

The initialized function only needs to be called once.

2.1.1 Initializing interface of initialization library

function	EEPTPU* init();
function	Initializing interface of initialization library. This initialization function needs to be called before using all class member functions.
parameter	Be without
return	Return EEPTPU class pointer
example	*EEPTPU *tpu = NULL; // Can be declared as a global variable* *if (tpu == NULL) tpu = tpu->init(); // Can be placed at the beginning of the main function for initialization*

2.1.2 Setting the interface type

function	int eeptpu_set_interface(int interface_type);
function	Setting the interface type of data interaction. After the init() function is executed, you need to call this function to set the interface type before loading the EEPTPU BIN file.
parameter	interface_type: The available interface types are：eepInterfaceType_SOC and eepInterfaceType_PCIE。Please select the appropriate interface type. （1）If the program is in the same onboard memory as the EEP-TPU, it is set to eepInterfaceType_SOC; （2）Set to eepInterfaceType_PCIE if the program interacts with EEP-TPU via PCIE on the host side. *enum {* *eepInterfaceType_NONE = 0,*

	eepInterfaceType_SOC = 1, *eepInterfaceType_PCIE,* *eepInterfaceType_EOF, /* dummy data. */* *};*
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	*tpu->eeptpu_set_interface(eepInterfaceType_SOC);*

2.1.3 Setting the PCIE device name

function	int eeptpu_set_interface_info_pcie(const char* dev_reg, const char* dev_h2c, const char* dev_c2h);
function	Set the PCIE device name.
parameter	**dev_reg**: The device name of the PCIE Register； **dev_h2c**: The device name of the PCIE host to card； **dev_c2h**: The device name of the PCIE card to host.
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	*tpu->eeptpu_set_interface_info_pcie("/dev/xdma0_user","/dev/xdma0_h2c_0","/dev/xdma0_c2h_0");*

2.1.4 Set the memory start address accessible by EEP-TPU

function	int eeptpu_set_tpu_mem_base_addr(unsigned long mem_base_addr);
function	Used for “eepInterfaceType_PCIE” interface mode. In “eepInterfaceType_SOC” mode, it can be left unconfigured.
parameter	**mem_base_addr**: The memory start address that the EEP-TPU can access and use. In PCIE mode, it is the starting memory address when reading and writing to the PCIE device on the EEP-TPU side.
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	*tpu->eeptpu_set_tpu_mem_base_addr(0x00000000);*

2.1.5 Set the information of the EEP-TPU register bank

function	int eeptpu_set_tpu_reg_zones(std::vector<struct EEPTPU_REG_ZONE>& regzones);
function	Configure the information about register banks for EEP-TPU modules. In “single-core SOC” mode, the default base address is already configured in the library file, and you do not need to call this function under normal circumstances; If the chip designer changes the default base address, the software developer needs to be told to call this function for configuration. For “multi-core” cases, this function needs to be called to configure the registers of each core.
parameter	**regzones**: Configuration information for the register bank of the EEP-TPU module. The information includes the ID of each EEP-TPU core, register base address, offset address, and register area size. Register Base Address: In “SOC interface” mode, the actual register address is passed in. In “PCIE interface” mode, the offset address of the EEP-TPU module relative to the overall register is passed. For example, if the overall register start address is 0x60000000, and the Core0 register of the EEP-TPU module starts from 0x60020000, then you can configure the register base address here to be 0x00020000.
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	**# SOC mode：** *vector<struct EEPTPU_REG_ZONE> regzones;* *struct EEPTPU_REG_ZONE zone;* *zone.core_id = 0;* *zone.addr = 0xA0000000;* *zone.size = 0x1000;* *regzones.push_back(zone);* *zone.core_id = 1;* *zone.addr = 0xA0040000;* *zone.size = 0x1000;* *regzones.push_back(zone);* *tpu->eeptpu_set_tpu_reg_zones(regzones);*

	# PCIE mode： *vector<struct EEPTPU_REG_ZONE> regzones;* *struct EEPTPU_REG_ZONE zone;* *zone.core_id = 0;* *zone.addr = 0x00020000;* *zone.size = 256*1024;* *regzones.push_back(zone);* *tpu->eeptpu_set_tpu_reg_zones(regzones);*
Remarks	# How to get the size of the register space in PCIE mode: **Method 1:** The first step is to load the Xilinx PCIE xdma driver; In the second step, execute the command in the command line window: lspci -v; In the third step, look for devices with "Kernel driver in use: xdma": *01:00.0 Memory controller: Xilinx Corporation Device 8028* *Subsystem: Xilinx Corporation Device 0007* *Flags: bus master, fast devsel, latency 0, IRQ 16* ***Memory at a1000000 (32-bit, non-prefetchable) [size=2M]*** ***Memory at a1200000 (32-bit, non-prefetchable) [size=64K]*** *Capabilities: <access denied>* *Kernel driver in use: xdma* Looking at the line where the Memory is located (line 1 usually represents the Bar0 of the PCIE device). Getting size of 2MB from this address, then converting it to decimal (2097152) or 16 decimal (0x200000) as a parameter into this function. **Method 2:** You can find the following log through the “dmesg” command after loading the Xilinx PCIE xdma driver: *BAR0 at 0xa1000000 mapped at 0x00000000512efc91, length=2097152(/2097152)* *BAR1 at 0xa1200000 mapped at 0x000000000ee6abb6, length=65536(/65536)* *config bar 1, pos 1.* *2 BARs: config 1, user 0, bypass -1.* Among them, the last sentence indicates that there are 2 bars, and config bar is Bar1, and user bar is Bar0; What we need to find is the user bar, which is Bar0. Looking at the address above, you can see that Bar0's length is 2097152.

2.1.6 Set the data memory base address of the EEP-TPU (2 base addresses)

Function	int eeptpu_set_base_address(unsigned long base0, unsigned long base1);
Function	Set the memory base address of the EEP-TPU. Related to the configuration of the boot core file, different development boards may require different memory configurations, which need to be configured according to the actual situation. Please consult the available base address when obtaining the boot file.
Parameter	base0: Base address 0 base1: Base address 1 **1. Description of algorithm memory space：** The EEP-TPU data used by each algorithm is divided into 4 categories: (1) Parameter data and algorithm instruction data (Parameter data: parameters used by each layer in the algorithm; Algorithm instruction data: the TPU instruction data of the algorithm. The parameter data and the algorithm instruction data are always unchanged during the TPU inference process.） (2) input data (3) output data (4) Temporary data, that is, the temporary data generated by TPU calculations during inference process. These four types of data can be assigned to the two different base addresses mentioned above. Call this function, the input and output data will be put together, using base address 0; the parameter data and algorithm instruction data with the temporary data will be put together, using base address 1. When the addresses set by base address 0 and base address 1 are the same, the four types of data mentioned above use the same memory space. **2. The order of memory data:** （1）When base0 equals base1： Input data, output data, parameter data, instruction data, temporary data. （2）When base0 is not equal to base1： Base0: Input data, output data； Base1: Parameter data, instruction data, temporary data.

	3. If multiple algorithms are used: When using "eeptpu_load_bin" API functions to load an algorithm, the parameter "fg_multi" is used to mark the loading of multiple algorithms. When the first algorithm is loaded, the "fg_multi" is set to 0； when subsequent algorithms are loaded, the "fg_multi" is set to 1. The two base addresses base0 and base1 are maintained by the library file. When a bin data is loaded, the base0 and base1 inside the library file are updated to the available memory addresses immediately after the previous algorithm. **4. The order of memory data in the case of multiple algorithms:** （1）When base0 equals base1： Algorithm 1 [input data, output data, parameter data, instruction data, temporary data]; Algorithm 2 [input data, output data, parameter data, instruction data, temporary data]; ...... （2）When base0 is not equal to base1： Base0: Algorithm 1 [input data, output data]; Algorithm 2 [input data, output data]; ...... Base1: Algorithm 1 [parameter data, instruction data, temporary data]; Algorithm 2 [parameter data, instruction data, temporary data]; ...... **5. Examples of algorithm usage:** Algorithm 1： *// eeptpu_set_base_address, only need to be configured once at the beginning.* *tpu1->eeptpu_set_base_address(0x30000000, 0x30000000);* *tpu1->eeptpu_load_bin(path_bin1, 0);* Algorithm 2: allocate the memory address after the algorithm 1, there is no need to configure the base address repeatedly. *tpu2->eeptpu_load_bin(path_bin2, 1);* "eeptpu_set_base_address" function only needs to be called once before loading the algorithm for the first time. When the first algorithm is loaded, the "fg_multi" parameter of the "eeptpu_load_bin" function is set to 0; and when the subsequent algorithm is loaded, "fg_multi" is set to 1. **6. Special operations and examples in the case of multiple algorithms:** In the case of multiple algorithms, users can also manually maintain the memory base address of each algorithm, as long as the memory data space allocated to each algorithm does not cover the data space of other algorithms. For example:

	Algorithm 1：The memory space size is 0xA00000. If you allocate from 0x30000000, the space is used until 0x30A00000. *tpu1->eeptpu_set_base_address(0x30000000, 0x30000000);* *tpu1->eeptpu_load_bin(path_bin1, 0);* Algorithm 2：Allocate space starting from 0x31000000, without covering algorithm 1. *tpu2->eeptpu_set_base_address(0x31000000, 0x31000000);* *tpu2->eeptpu_load_bin(path_bin2, 0);* "eeptpu_set_base_address" function needs to be called every time. The "fg_multi" parameter of "eeptpu_load_bin" function is 0, which indicates the base address is set by the previous "eeptpu_set_base_address" function. **7. Acquisition of memory space size:** You can use the "eeptpu_get_memory_used_size( )" API function to get the memory space occupied by the algorithm. If there are multiple algorithms, the total amount of memory space required is the sum of the space obtained by each algorithm. The data used by each algorithm is limited to the memory address space allocated to it, and no additional memory address space is used dynamically. For example： *unsigned long memsize1 = tpu1->eeptpu_get_memory_used_size();* *int MBytes1 = memsize1 / (1024*1024);* *int KBytes1 = memsize1 % (1024*1024);* *unsigned long memsize2 = tpu2->eeptpu_get_memory_used_size();* *unsigned long memtotal = memsize1 + memsize2;*
Return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
Example	*tpu->eeptpu_set_base_address(0x30000000, 0x30000000);*

2.1.7 Setting the Memory Base Address of EEP-TPU Data (4 Base Addresses)

function	int eeptpu_set_base_address(unsigned long base_par, unsigned long base_in, unsigned long base_out, unsigned long base_tmp);
function	Set the memory base address of the EEP-TPU, and you can set four different types of memory base addresses. Related to the configuration of the boot core file, different

	development boards may require different memory configurations, which need to be configured according to the actual situation. Please consult the available base address when obtaining the boot file.
parameter	base_par: Algorithm parameter data(including algorithm instruction data); base_in: Input data; base_out: Output data; base_tmp: Temporary data. **Description of algorithm memory space：** 1. The EEP-TPU data is divided into 4 categories: (1) Parameter data and algorithm instruction data (Parameter data: parameters used by each layer of the algorithm; Algorithm instruction data: the TPU instruction data of the algorithm. The parameter data and the algorithm instruction data are always unchanged during the TPU inference process.） (2) Input data; (3) Output data; (4) Temporary data, that is, the temporary data generated by TPU calculations during inference process. These four types of data can be configured separately into the base addresses mentioned above. In the 4 base addresses, there can be the same base address. If the same base address exists, it means that the corresponding data is stored in the memory space at the beginning of the base address. In this case, the order of the data is:parameter data, input data, output data, temporary data. 2. The use of multiple algorithms is consistent with the instructions in Section 2.1.6.
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	*tpu->eeptpu_set_base_address(0x60000000, 0x60000000, 0x60000000, 0x60000000);*

2.1.8 Load EEP-TPU BIN files

function	int eeptpu_load_bin(const char* path_bin, int fg_multi=0);
function	Load the bin file generated by the eeptpu compiler.

	NOTE： An EEPTPU object can only load one EEPTPU BIN file. If you need to use multiple EEPTPU BIN files, you need to define multiple EEPTPU objects.
parameter	**path_bin**: The path to the bin file; **fg_multi**: 是否已加载多个 bin 文件。Whether multiple bin files have been loaded.When calling this function for the first time in the program, the “fg_multi” must be set to 0; if multiple bin files need to be loaded in a program, the “fg_multi” starting from the second and subsequent must be set to 1. In the case of multiple cores, “fg_multi” is set up in the same way.
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	*tpu->eeptpu_load_bin(path_bin);* To load multiple bin files: *tpu1->eeptpu_load_bin(path_bin); // If the second parameter is not filled in, the default = 0* *tpu2->eeptpu_load_bin(path_bin, 1);* *tpu3->eeptpu_load_bin(path_bin, 1);*

2.1.9 Configuring algorithm jumping

function	int eeptpu_jump_update(EEPTPU* alg2);
function	EEP-TPU supports jumping from algorithm 1 to algorithm 2。The output of algorithm 1 is taken as the input to algorithm 2. When applying algorithm jumps, both algorithm 1 and algorithm 2 need to be initialized, and then call this function to set the jumping. During inference, you only need to call the inference of algorithm 1. The TPU will automatically jump to algorithm 2. After the inference of algorithm 1 is completed, and the result data returned by the inference is also the inference result of algorithm 2.
parameter	**alg2：** 2nd algorithm
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	*EEPTPU *tpu1 = NULL;* *EEPTPU *tpu2 = NULL;* *>> Initialize tpu1 and tpu2*

	//jumping from tpu1 to tpu2. *ret = tpu1->eeptpu_jump_update(tpu2);* *>> tpu1 writes the input data* *tpu1->eeptpu_forward(result);* *>> Only TPU1 inference is required. After the inference of algorithm 1 is completed, and the result data returned by the inference is also the inference result of algorithm 2.*

2.1.10 Get the memory usage size of the algorithm

function	unsigned long eeptpu_get_memory_used_size();
function	Get the memory space used by the algorithm.
parameter	without
return	The number of bytes occupied by memory.
example	*unsigned long memsize = tpu->eeptpu_get_memory_used_size();* *int MBytes = memsize / (1024*1024);* *int KBytes = memsize % (1024*1024);*

2.1.11 Getting the version of the library file

function	char* eeptpu_get_lib_version();
function	Getting the version information of the library file.
parameter	without
return	Version string
example	*char* ptr = tpu->eeptpu_get_lib_version();* *printf("EEPTPU library version: %s\n", ptr);*

2.1.12 Getting EEP-TPU hardware version

function	char* eeptpu_get_tpu_version();
function	Getting EEP-TPU hardware version information.
parameter	without
return	Version string
example	*char* ptr = tpu->eeptpu_get_tpu_version();* *printf("EEPTPU hardware version: %s\n", ptr);*

2.1.13 Getting EEP-TPU hardware configuration information

function	char* eeptpu_get_tpu_info();
function	Getting EEP-TPU hardware configuration information.
parameter	Without
return	String
example	*char* ptr = tpu->eeptpu_get_tpu_info();* *printf("EEPTPU hardware info : %s\n", ptr);*

2.1.14 Getting EEP-TPU BIN file extension information

function	char* eeptpu_get_extinfo();
function	Getting the extended information string customized by the EEP-TPU bin file. The string is passed in at compiler time via the command-line parameter "*--extinfo*", which can be customized by the user and parsed by itself.
parameter	Without
return	String
example	char* ptr = tpu->eeptpu_get_extinfo(); printf("extinfo: %s\n", ptr);

2.2 Input data

2.2.1 Wait for input buffer to become writable

function	int eeptpu_wait_input_writable(unsigned int timeout_ms = 2000);
function	Wait for the input buffer to become writable. When the data in the input buffer has not been exhausted, the input buffer cannot be rewritten, otherwise the input data will be confused.
parameter	**timeout_ms**: The timeout for waiting, in milliseconds. (Default value is 2000ms)
return	0: Success. You can continue to write data. < 0: Failed. (Error codes refer to Annex 1). If the data in the current data buffer has not been

	exhausted, we recommend that you set a longer timeout period. If forced to write, the input data could be confused and affect the inference result.
example	*ret = tpu->eeptpu_wait_input_writable(2000);*

2.2.2 Settings for input data (single input case)

function	int eeptpu_set_input(void* input_data, int dim1, int dim2, int dim3, int mode = 0, int data_type = DType_FP32);
function	In cases where the neural network is a single input, set the input data for the network.
parameter	**1. input_data**： For the preprocessed input data, its dimensions need to be the same as neural networks. Preprocessing refers to operations such as resize, sub-meaning, and normalization. In the case of image data, the mean and normalization operations can be automatically processed by the library file. That is, the user could not do the mean and normalization in the preprocessing. If the input is image data: It can be a data pointer in Mat format of Opencv. **2. dim1、dim2、dim3**：Dimensions of input data. If the input is image data: dim1, dim2, and dim3 refer to the number of channels, height, and width of the image, respectively. **3. Mode**：Pattern of input data. Mode=0：The default value of mode is 0. In this case, the meaning and normalization can be processed through the library file. This is for the case where the input data is an image. The mean value is set when compiling neural networks with the eeptpu compiler, and when this function is called, the image data can be input and the mean value can be calculated automatically. Mode=1：In this case, no meaning and normalization calculations are performed inside the library file. Therefore, before calling this function, the user needs to do all the preprocessing operations on the input data. If the mode is equal to 0 or 1, the library file converts the data into a default format that can be recognized by the TPU before the input data is written to memory. Mode=2：The original input data is written directly to the memory for use by the TPU without any conversion. This is for the case where the input data is “pack” mode

	data_type: The data type of the input data, suitable for mode=1. The default is the float32 type. Supported data types are:FP32/FP16/INT8/INT16/INT32/UINT8。 The dimension of the input data of the neural network can be obtained by *tpu->input_shape[n]* after loading the bin file. N value range: 0~3; respectively indicate batch_size, C, H, W. Note: **The batch_size is always 1.**
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	*cv::Mat cvimg_resized;* *>> read&process image to cvimg_resized* *tpu->eeptpu_set_input(cvimg_resized.data,tpu->input_shape[1],tpu->input_shape[2],tpu->input_shape[3]);*

2.2.3 Setting for input data (multi-input case)

function	int eeptpu_set_input(int input_id, void* input_data, int dim1, int dim2, int dim3, int mode = 0, int data_type = DType_FP32);
function	In cases where the neural network is multiple inputs, set the input data for the network.
parameter	**input_id**: Input the index number of the data. This index number can be printed when the compiler compiles the bin file. **Other parameters:** Consistent with the description of the single input case.
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	*cv::Mat cvimg_resized_0;* *>> read&process image to cvimg_resized_0* *tpu->eeptpu_set_input(0, cvimg_resized.data,c0, h0, w0);* *cv::Mat cvimg_resized_1;* *>> read&process image to cvimg_resized_1* *tpu->eeptpu_set_input(1, cvimg_resized.data,c1, h1, w1);*

2.2.4 Getting information about input data (multi-input case)

function	int eeptpu_get_input_info(std::vector<struct NET_INPUT_INFO>& input_info);

function	Getting information about the input data. Suitable for single and multiple input situations.
parameter	**input_info**：The information about the input data. “NET_INPUT_INFO” structure data. The information included are: (1) input_id: Index number of the input data. (For single input, the index number is 0) (2) c/h/w: 3 dimensions of the input data. (3) name: Name of the input data. Corresponds to the name of the input data printed during compiler compilation. You can also view the input data name in the “Netron” Network Visualizer. (4) mean/norm: The mean and normalized configuration corresponding to this input data. (5) pack_type: The input data type of “pack” when in “Pack” mode. (6) pack_out_c/pack_out_h/pack_out_w：The dimension after the pack operation in “Pack” mode.
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	*std::vector<struct NET_INPUT_INFO> inputs_info;* *tpu->eeptpu_get_input_info(inputs_info);*

2.2.5 Getting the mean and normalized configuration of input data (single input case)

function	void eeptpu_get_mean_norm(std::vector<float>& get_mean, std::vector<float>& get_norm);
function	Getting the mean and normalized configuration of input data.
parameter	get_mean: Mean data. Corresponds to the configuration of the compiler's parameter “--mean”. get_norm: Normalize data. Corresponds to the configuration of the compiler's parameter “--norm”.
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	*vector<float> net_mean;* *vector<float> net_norm;* *tpu->eeptpu_get_mean_norm(net_mean, net_norm);*

2.3 Inference

2.3.1 Setting the wait timeout for the end of inference

function	int eeptpu_set_forward_timeout(unsigned int ms);
function	When performing an inference function, there is a wait timeout waiting for the end of inference; The default value is 20000 milliseconds. Users can configure a suitable waiting time.
parameter	**ms**: timeout (in milliseconds)
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	*int ret = tpu-> eeptpu_set_forward_timeout(3000);*

2.3.2 Forward reasoning

function	int eeptpu_forward(std::vector<struct EEPTPU_RESULT>& result);
function	Image reasoning.
parameter	**result**: inference result *struct EEPTPU_RESULT* *{* *float* data; // Final data* *int shape[4]; // Dimension of output data（shape[0]=1）* *};*
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	*std::vector<struct EEPTPU_RESULT> results;* *int ret = tpu->eeptpu_forward(results);*

2.3.3 Get the inference time of the EEP-TPU hardware part

function	unsigned int eeptpu_get_tpu_forward_time();
function	Get the inference time of the EEP-TPU hardware part. (Unit: microseconds)

parameter	Without.
return	The inference time of the EEP-TPU hardware part. (Unit: microseconds)
example	*unsigned int hwus = tpu->eeptpu_get_tpu_forward_time();* *printf("EEPTPU hw cost: %.3f ms\n", (float)hwus/1000);*

2.4 Abort

2.4.1 Abort EEP-TPU operation

function	void eeptpu_terminate(bool b_term);
function	EEP-TPU “abort operation” or “Cancel ‘abort operation’ ”.
parameter	**b_term**: true- abort operation; false- Cancel the aborted running state.
return	Without.
example	*tpu->eeptpu_terminate(true);* Example: After capturing an interrupt signal such as Ctrl+C in the application, this function can be called to abort and exit the EEP-TPU inference function in time. Alternatively, in a multithreaded environment, thread 1 constantly does TPU inference in a loop; Thread 2 can call this function to pass true to abort the inference of the TPU, and then pass false to end the abort state the next time inference is required.

2.4.2 Turn off EEP-TPU

function	void eeptpu_close();
function	Turn off EEP-TPU。
parameter	Without.
return	Without.
example	*tpu->eeptpu_close();* This function is called to turn off the EEP-TPU when the application shuts down.

2.5 Memory Mapping

2.5.1 Memory Mapping

function	void* eeptpu_memory_map(unsigned int addr, unsigned int len);
function	Map a piece of memory and get its pointer.
parameter	**Addr**: The starting address of the memory to be mapped. **Len**: Memory size to be mapped.
return	Returns the mapped memory pointer. If NULL, the mapping failed.
example	*unsigned char* ptr = tpu-> eeptpu_memory_map(0x65000000, 0x1000);*

2.5.2 Unmapped memory

function	int eeptpu_memory_unmap(void* start, unsigned int len);
function	Unmapped a memory segment.
parameter	**Start**: eeptpu_memory_map The memory pointer returned by the function. **Len**: Mapped memory size.
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	*unsigned char* ptr = tpu->eeptpu_memory_map(0x65000000, 0x1000);* *int ret = tpu->eeptpu_memory_unmap(ptr, 0x1000);*

2.5.3 Read memory

function	int eeptpu_mem_rd(unsigned char* buf, unsigned long addr, unsigned int len);
function	Read the data of a segment of memory.
parameter	**Buf**: Data pointer, where read memory data is stored. The user needs to set aside a large enough memory space for this function before calling it. **addr**：Memory address. **Len**: Length of memory bytes.
return	0: Success；

	< 0: Failed. (Please refer to the error code in Appendix 1)
example	*unsigned char* buf = (unsigned char*)malloc(0x1000);* *ret = tpu->eeptpu_mem_rd(buf, 0x60000000, 0x1000);*

2.5.4 Write memory

function	int eeptpu_mem_wr(unsigned char* buf, unsigned long addr, unsigned int len);
function	Write data to memory.
parameter	**Buf**: Data pointer, where the data to be written to memory is stored. **Addr**: Memory address. **Len:** The length of bytes in which data is written.
return	0: Success; < 0: Failed. (Please refer to the error code in Appendix 1)
example	*unsigned char* buf = (unsigned char*)malloc(0x1000);* *>> Padding buf data* *ret = tpu->eeptpu_mem_wr(buf, 0x60000000, 0x1000);*

2.6 Parameter multiplexing function

For multicore EEP-TPUs, when an algorithm (bin file) wants to use multiple cores at the same time, you can use the parameter multiplexing function described in this section.

In general, when using single or multiple cores for inference, each core may be responsible for a different algorithm. In the process of initialization, each core must carry out its own initialization (such as configuring the memory base address of each core, register information, load bin file, etc.), especially the load bin file, which will write the algorithm data in the bin to memory and occupying a certain memory space.

When multiple cores need to use the same algorithm, in order to save memory space and avoid wasting memory space caused by loading the same bin file multiple times, you can use the parameter multiplexing function in this section. So that you only need to load the parameter data in this bin file once, and the parameter data, input data, output data, and temporary data can be customized and flexibly allocated memory address space, so that the algorithm can be configured more flexibly.

2.6.1 Loading algorithm configuration information

function	int eeptpu_nn_load_info(const char* path_bin);
function	Load the neural network algorithm configuration information in the bin file, and will not write the algorithm parameters and instruction information to memory.
parameter	**path_bin**: The path of bin file.
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	#define ALG_NUM 2 // count of used bin files EEPTPU* alg[ALG_NUM]; for (unsigned int i = 0; i < ALG_NUM; i++) { printf("\nReading alg[%d] info...\n", i); alg[i] = alg[i]->init(); ret = alg[i]->eeptpu_set_interface(eepInterfaceType_SOC); if (ret < 0) return ret; ret = alg[i]->eeptpu_nn_load_info(bins[i].c_str()); if (ret < 0) { printf("Load info fail, ret=%d\n", ret); return ret; } }

2.6.2 Loading algorithm parameter data

function	int eeptpu_nn_load_data(unsigned long addr, unsigned int len, int flag);
function	Load the neural network algorithm parameters or algorithm instructions in the bin file and write them to memory.
parameter	**Addr**: Memory address. **Len**: Data length. **Flag**: flag bit, 1 means to load algorithm parameter data, 2 means to load algorithm instruction data.

return	0: Success; < 0: Failed. (Please refer to the error code in Appendix 1)
example	*ret = alg[0]->eeptpu_nn_load_data(0x60000000, 0x18000, 1);*

2.6.3 Set the memory base address of EEP-TPU data

function	int eeptpu_set_base_address(struct NET_MEM_ADDR mem_base);
function	Set the memory start address where the data is stored.
parameter	**mem_base**: NET_MEM_ADDR structure data. Five types of data can be configured to store the base address: parameters, algorithm instructions, input, output, and temporary data.
return	0: Success; < 0: Failed. (Please refer to the error code in Appendix 1)
example	*EEPTPU* cores[2];* *vector<struct NET_MEM_ADDR> cores_membase;* *>> Initialization cores and cores_membase* *ret = cores[0]->eeptpu_set_base_address(cores_membase[0]);*

2.6.4 Copying data related to neural networks

function	int eeptpu_net_copy(EEPTPU* alg);
function	Copy information about another EEP-TPU object that has been initialized and loaded with data.
parameter	**Alg**: The EEP-TPU object pointer has been initialized and loaded with data.
return	0: Success; < 0: Failed. (Please refer to the error code in Appendix 1)
example	*ret = cores[0]->eeptpu_net_copy(alg[1]);*

2.6.5 Get the memory space size of each data

function	struct NET_MEM_SIZE eeptpu_get_memory_zone_size();
function	Gets the memory space size of various types of data for the currently loaded algorithm.
parameter	Without.

return	NET_MEM_SIZE structure data, including the memory size of 5 types of data: parameters, algorithm instructions, input, output, and temporary data.
example	vector<struct NET_MEM_SIZE> memsize; for (unsigned int i = 0; i < ALG_NUM; i++) { memsize[i] = alg[i]->eeptpu_get_memory_zone_size(); printf("alg[%d] memsize: par 0x%08x; alg 0x%08x; in 0x%08x; out 0x%08x; tmp 0x%08x\n", i, memsize[i].par, memsize[i].alg, memsize[i].in, memsize[i].out, memsize[i].tmp); }

2.6.6 Getting the actual memory address of various types of data

function	int eeptpu_get_memory_zone_addr(int flag);
function	Get the actual memory address of various types of data.
parameter	**flag**：Flag bits; idxPar, idxIn, idxTmp, idxOut, idxAlg defined in the eeptpu.h header file.
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	unsigned long addr_par = tpu->eeptpu_get_memory_zone_addr(idxPar);

2.6.7 Getting the memory footprint of various types of data

function	unsigned int eeptpu_get_memory_zone_size(int flag);
function	Get the number of memory bytes occupied by various types of data.
parameter	**flag**：flag bit; idxPar, idxIn, idxTmp, idxOut, idxAlg defined in the eeptpu.h header file.
return	0: Success； < 0: Failed. (Please refer to the error code in Appendix 1)
example	unsigned int len_par = tpu->eeptpu_get_memory_zone_size(idxPar);

3. Multiple neural networks are used interchangeably in the same program

The EEP-TPU API supports the alternating use of multiple neural networks (eeptpu bin) in the same program.

Initialize the sample code:

```
EEPTPU *tpu1 = NULL;
EEPTPU *tpu2 = NULL;
EEPTPU *tpu3 = NULL;
char path_bin1[ ] = (char*)“./eeptpu1.pub.bin”;
char path_bin2[ ] = (char*)“./eeptpu2.pub.bin”;
char path_bin3[ ] = (char*)“./eeptpu3.pub.bin”;
int eeptpu_init(int interface_type)
{
    int ret;
    if (tpu1 == NULL) tpu1 = tpu1->init();
    if (tpu2 == NULL) tpu2 = tpu2->init();
    if (tpu3 == NULL) tpu3 = tpu3->init();
    // Configure the 1st EEPTPU object
    ret = tpu1->eeptpu_set_interface(interface_type);
    if (ret < 0) return ret;
    ret = tpu1->eeptpu_load_bin(path_bin1);
    if (ret < 0) return ret;
    // Configure the 2nd EEPTPU object
    ret = tpu2->eeptpu_set_interface(interface_type);
    if (ret < 0) return ret;
    ret = tpu2->eeptpu_load_bin(path_bin2, 1);
    if (ret < 0) return ret;
```

```
    // Configure the 3rd EEPTPU object
    ret = tpu3->eeptpu_set_interface(interface_type);
    if (ret < 0) return ret;
    ret = tpu3->eeptpu_load_bin(path_bin3, 1);
    if (ret < 0) return ret;
    return 0;
}
```

Attachment 1 - List of Error Codes

error code	Error description
0	success
-1	fail
-2	Parameter error
-3	Unsupported operations
-4	File opening failed
-5	File read failed
-6	File write error
-7	Memory allocation failed
-8	timeout
-11	Failed to load Bin file
-12	EEP-TPU initialization failed
-13	Wrong image type
-14	Unsupported pixel types
-15	Mean setting failed
-16	Incorrect blob input
-17	Wrong blob format
-18	Wrong blob size
-19	Wrong blob data
-20	Bad blob output
-21	Memory address error
-22	Memory data write failed
-23	Memory data read failed

-24	Memory mapping failed
-25	Device failed to open
-26	Device not initialized
-27	EEP-TPU interface type setting failed
-28	EEP-TPU interface initialization failed
-29	EEP-TPU interface operation failed
-40	The version of the EEP-TPU Bin file is outdated, please regenerate the Bin file with the new version compiler
-41	The EEP-TPU library file version is outdated, please use the new version of the library file
-42	The hardware version of EEP-TPU is outdated, please upgrade EEP-TPU