4.4 神经网络加速器

一、实验目的

本节课主要学习 K210 芯片中的神经网络加速器 KPU 的功能。

二、实验准备

1.实验元件

K210 芯片中的神经网络加速器 KPU

2.元件特性

KPU 是通用神经网络处理器，内置卷积、批归一化、激活、池化运算单元，可以对人脸或物体进行实时检测，具体特性如下：

- 支持主流训练框架按照特定限制规则训练出来的定点化模型
- 对网络层数无直接限制，支持每层卷积神经网络参数单独配置，包括输入输出通道数目、输入输

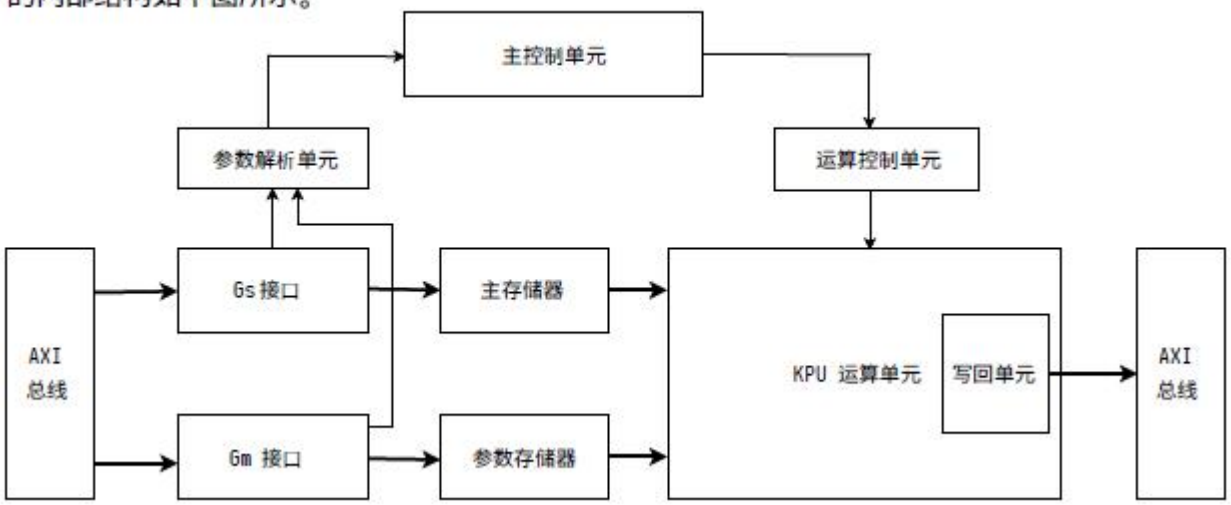
出行宽列高

- 支持两种卷积内核 1x1 和 3x3
- 支持任意形式的激活函数
- 实时工作时最大支持神经网络参数大小为 5.5MiB 到 5.9MiB
- 非实时工作时最大支持网络参数大小为（Flash 容量-软件体积）

工况	最大定点模型大小（MiB）	量化前浮点模型大小（MiB）
实时（≥30fps）	5.9	11.8

非实时（＜10fps）	与 flash 容量相关	与 flash 容量相关

KPU 的内部结构如下图所示。



4. SDK 中对应 API 功能

对应的头文件 kpu.h

为用户提供以下接口：

- kpu_task_init (0.6.0 以后不再支持，请使用 kpu_single_task_init)：初始化 kpu 任务句柄，该函数具体实现在 model compiler 生成的 gencode_output.c 中。
- kpu_run (0.6.0 以后不再支持，请使用 kpu_start)：启动 KPU，进行 AI 运算。
- kpu_get_output_buf (0.6.0 以后不再支持)：获取 KPU 输出结果的缓存。
- kpu_release_output_buf (0.6.0 以后不再支持)：释放 KPU 输出结果缓存。
- kpu_start：启动 KPU，进行 AI 运算。
- kpu_single_task_init：初始化 kpu 任务句柄。
- kpu_single_task_deinit：注销 kpu 任务。
- kpu_model_load_from_buffer：解析 kmodel 并初始化 kpu 句柄。
- kpu_load_kmodel：加载 kmodel，需要与 nncase 配合使用。
- kpu_model_free：释放 kpu 资源。
- kpu_get_output：获取 KPU 最终处理的结果。

• kpu_run_kmodel：运行 kmodel。

三、实验原理

KPU 可以接收图像的数据，然后经过神经网络卷积计算，返回计算后的数据，LCD 显示输出。

四、实验过程

1. 首先是 LCD、摄像头和按键的硬件引脚和软件 GPIO 映射关系。

```
/*****************************HARDWARE-PIN*********************************
// 硬件IO口，与原理图对应
#define PIN_LCD_CS              (36)
#define PIN_LCD_RST             (37)
#define PIN_LCD_RS              (38)
#define PIN_LCD_WR              (39)

// camera
#define PIN_DVP_PCLK            (47)
#define PIN_DVP_XCLK            (46)
#define PIN_DVP_HSYNC           (45)
#define PIN_DVP_PWDN            (44)
#define PIN_DVP_VSYNC           (43)
#define PIN_DVP_RST             (42)
#define PIN_DVP_SCL             (41)
#define PIN_DVP_SDA             (40)

#define PIN_KEYPAD_MIDDLE       (2)

/*****************************SOFTWARE-GPIO********************************
// 软件GPIO口，与程序对应
#define LCD_RST_GPIONUM         (0)
#define LCD_RS_GPIONUM          (1)

#define KEYPAD_MIDDLE_GPIONUM   (2)

/*****************************FUNC-GPIO************************************
// GPIO口的功能，绑定到硬件IO口
#define FUNC_LCD_CS             (FUNC_SPI0_SS3)
#define FUNC_LCD_RST            (FUNC_GPIOHS0 + LCD_RST_GPIONUM)
#define FUNC_LCD_RS             (FUNC_GPIOHS0 + LCD_RS_GPIONUM)
#define FUNC_LCD_WR             (FUNC_SPI0_SCLK)

#define FUNC_KEYPAD_MIDDLE      (FUNC_GPIOHS0 + KEYPAD_MIDDLE_GPIONUM)
```

```
void hardware_init(void)
{
    /* 按键 */
    fpioa_set_function(PIN_KEYPAD_MIDDLE, FUNC_KEYPAD_MIDDLE);

    /* LCD */
    fpioa_set_function(PIN_LCD_CS,  FUNC_LCD_CS);
    fpioa_set_function(PIN_LCD_RST, FUNC_LCD_RST);
    fpioa_set_function(PIN_LCD_RS,  FUNC_LCD_RS);
    fpioa_set_function(PIN_LCD_WR,  FUNC_LCD_WR);

    // DVP camera
    fpioa_set_function(PIN_DVP_RST,   FUNC_CMOS_RST);
    fpioa_set_function(PIN_DVP_PWDN,  FUNC_CMOS_PWDN);
    fpioa_set_function(PIN_DVP_XCLK,  FUNC_CMOS_XCLK);
    fpioa_set_function(PIN_DVP_VSYNC, FUNC_CMOS_VSYNC);
    fpioa_set_function(PIN_DVP_HSYNC, FUNC_CMOS_HREF);
    fpioa_set_function(PIN_DVP_PCLK,  FUNC_CMOS_PCLK);
    fpioa_set_function(PIN_DVP_SCL,   FUNC_SCCB_SCLK);
    fpioa_set_function(PIN_DVP_SDA,   FUNC_SCCB_SDA);

    // 使能SPI0和DVP
    sysctl_set_spi0_dvp_data(1);
}
```

2. 设置摄像头和显示器接口需要的电平电压为 1.8V。

```
static void io_set_power(void)
{
    /* Set dvp and spi pin to 1.8V */
    sysctl_set_power_mode(SYSCTL_POWER_BANK6, SYSCTL_POWER_V18);
    sysctl_set_power_mode(SYSCTL_POWER_BANK7, SYSCTL_POWER_V18);
}
```

3. 设置系统时钟，初始化系统中断，并使能系统全局中断。

```
/* 设置系统时钟和DVP时钟 */
sysctl_pll_set_freq(SYSCTL_PLL0, 800000000UL);
sysctl_pll_set_freq(SYSCTL_PLL1, 300000000UL);
sysctl_pll_set_freq(SYSCTL_PLL2, 45158400UL);
uarths_init();

/* 系统中断初始化 */
plic_init();
/* 使能系统全局中断 */
sysctl_enable_irq();
```

4. 初始化显示屏，并显示图片一秒。

```
/* 初始化显示屏，并显示一秒图片 */
printf("LCD init\r\n");
lcd_init();
lcd_draw_picture_half(0, 0, 320, 240, gImage_logo);
sleep(1);
```

5. 初始化摄像头。

```
/* ov摄像头初始化 */
int OV_type;
OV_type=OVxxxx_read_id();
/* 初始化摄像头 */
if(OV_type == OV_9655)
{
    ov9655_init();
}
else if(OV_type == OV_2640)
{
    ov2640_init();
}
else
{
   return 0;//打不开摄像头，结束
}
```

6. 初始化按键并且设置按键中断回调。

```
/* 按键中断回调 */
int key_irq_cb(void *ctx)
{
    key_flag = 1;
    key_state = gpiohs_get_pin(KEYPAD_MIDDLE_GPIONUM);
    return 0;
}

/* 初始化按键 */
void init_key(void)
{
    // 设置按键的GPIO模式为上拉输入
    gpiohs_set_drive_mode(KEYPAD_MIDDLE_GPIONUM, GPIO_DM_INPUT_PULL_UP);
    // 设置按键的GPIO电平触发模式为上升沿和下降沿
    gpiohs_set_pin_edge(KEYPAD_MIDDLE_GPIONUM, GPIO_PE_BOTH);
    // 设置按键GPIO口的中断回调
    gpiohs_irq_register(KEYPAD_MIDDLE_GPIONUM, 1, key_irq_cb, NULL);
}
```

7. KPU 初始化并打印剩余内存。

```
/* kpu初始化 */
kpu_task_t task;
conv_init(&task, CONV_3_3, conv_data);

printf("KPU TASK INIT, FREE MEM: %ld\r\n", get_free_heap_size());
printf("Please press the keypad to switch mode\r\n");
```

8. 等待摄像头传输数据完成，kpu 开始运算摄像头采集的数据，并输出到 g_ai_buf_out 中。

```
while (g_dvp_finish_flag == 0)
    ;
/* 开始运算 */
conv_run(&task, g_ai_buf_in, g_ai_buf_out, kpu_done);
```

```
/* KPU完成 */
static int kpu_done(void *ctx)
{
    g_ai_done_flag = 1;
    return 0;
}
```

9. 等待 KPU 计算完成，就把数据转化成 RGB565 格式，因为显示器不支持 RGB888 格式。

```
while (!g_ai_done_flag)
    ;
g_ai_done_flag = 0;
g_dvp_finish_flag = 0;
/* 转化成LCD支持的RGB565格式 */
rgb888_to_565(g_ai_buf_out, g_ai_buf_out + 320 * 240, g_ai_buf_out + 320 * 240 * 2,
    (uint16_t *)g_display_buf, 320 * 240);
```

```
/* 转化图像数据格式，因为摄像头输出到AI的是RGB888格式，而显示屏需要RGB565格式 */
void rgb888_to_565(uint8_t *src_r, uint8_t *src_g, uint8_t *src_b, uint16_t *dst, uint32_t
{
    uint32_t i;
    for (i = 0; i < len; i += 2)
    {
        dst[i] = (((uint16_t)(src_r[i + 1] >> 3)) << 11) +
            (((uint16_t)src_g[i + 1] >> 2) << 5) +
            (((uint16_t)src_b[i + 1]) >> 3);
        dst[i + 1] = (((uint16_t)(src_r[i] >> 3)) << 11) +
            (((uint16_t)src_g[i] >> 2) << 5) +
            (((uint16_t)src_b[i]) >> 3);
    }
}
```

10. 此时我们需要在左上角写上每个对应的模式，默认是原始 origin。

```
/* 左上角显示模式 */
void draw_text(void)
{
    char string_buf[8 * 16 * 2 * 16]; //16个字符
    char title[20];

    switch (demo_index)
    {
    case 0:
        sprintf(title, " origin ");
        lcd_ram_draw_string(title, (uint32_t *)string_buf, BLUE, BLACK);
        lcd_ram_cpyimg((char *)g_display_buf, 320, string_buf, strlen(title) * 8, 16, 0,
        break;
    case 1:
        sprintf(title, "  edge  ");
        lcd_ram_draw_string(title, (uint32_t *)string_buf, BLUE, BLACK);
        lcd_ram_cpyimg((char *)g_display_buf, 320, string_buf, strlen(title) * 8, 16, 0,
        break;
    case 2:
        sprintf(title, " sharp  ");
        lcd_ram_draw_string(title, (uint32_t *)string_buf, BLUE, BLACK);
        lcd_ram_cpyimg((char *)g_display_buf, 320, string_buf, strlen(title) * 8, 16, 0,
        break;
    case 3:
        sprintf(title, "relievos");
        lcd_ram_draw_string(title, (uint32_t *)string_buf, BLUE, BLACK);
        lcd_ram_cpyimg((char *)g_display_buf, 320, string_buf, strlen(title) * 8, 16, 0,
        break;

    default:
        break;
    }
```

11. 最后是把数据显示到 LCD 上。

```
/* 左上角写字母提示是哪个模式 */
draw_text();
/* 显示图像 */
lcd_draw_picture(0, 0, 320, 240, g_display_buf);
```

12. 那么怎么切换显示的模式呢？那就是通过按键的中断修改按键的状态来切换模式的。

```
if (key_flag) //使用按键选择的卷积核
{
    if (key_state == 0) //按下
    {
        msleep(20); //延迟去抖
        key_flag = 0;
        demo_index = (demo_index + 1) % 4;
        memcpy((void *)conv_data, (void *)(conv_data_demo[demo_index]),
            3 * 3 * 3 * 3 * sizeof(float));
        conv_init(&task, CONV_3_3, conv_data);
    }
    else //弹起
    {
        msleep(20); //延迟去抖
        key_flag = 0;
    }
}
```

13.编译调试，烧录运行

把本课程资料中的 kpu 复制到 SDK 中的 src 目录下，然后进入 build 目录，运行以下命令编译。

cmake .. -DPROJ=kpu -G "MinGW Makefiles"

make

```
[ 89%] Linking C executable kpu
Generating .bin file ...
[100%] Built target kpu
PS C:\K210\SDK\kendryte-standalone-sdk-develop\build>
```

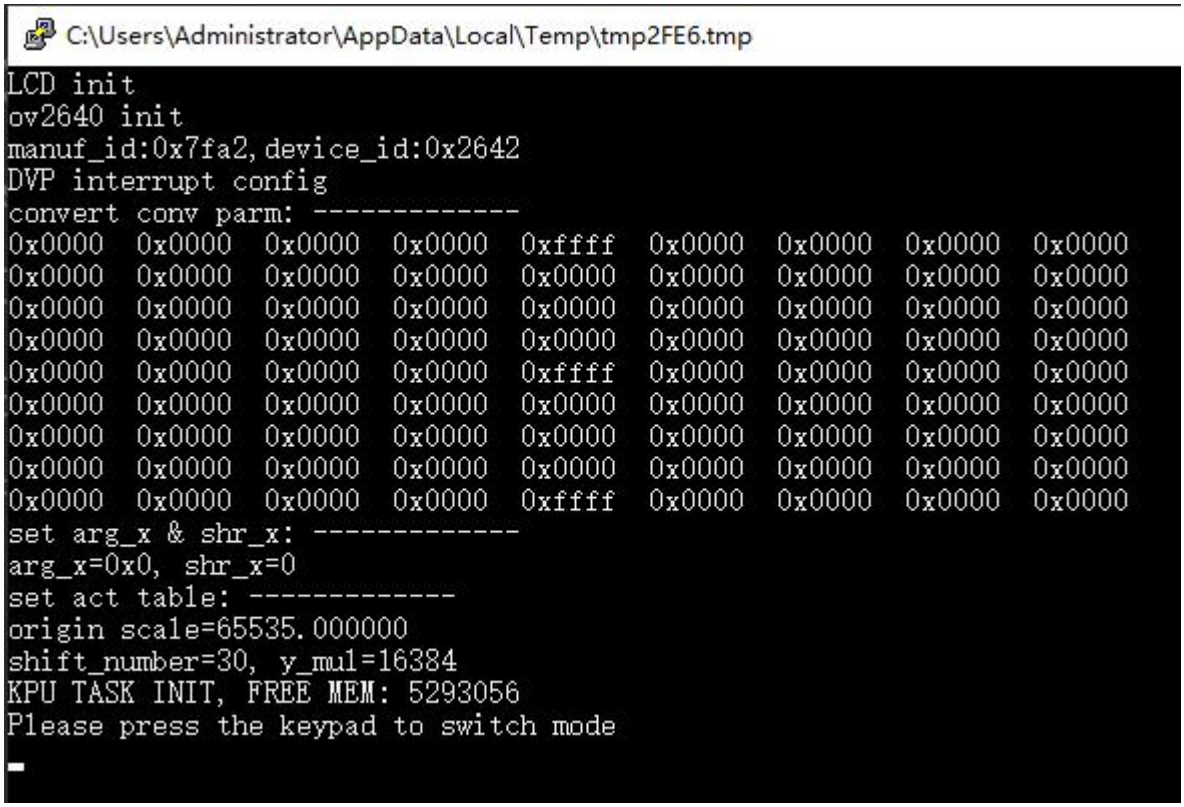
编译完成后，在 build 文件夹下会生成 kpu.bin 文件。

使用 type-C 数据线连接电脑与 K210 开发板，打开 kflash，选择对应的设备，再将程序固件烧录到 K210 开发板上。

五、实验现象

烧录固件完成后，系统会自动弹出一个终端窗口，并且打印一些初始化的信

息，此时我们看显示器的已经显示了摄像头当前采集的画面，而且左上角还有一个‘origin’的单词，当我们按下 keypad 中间的键时，模式切换，LCD 显示的画面会变化，除了原始画面，还有其他三种模式可以显示，每按一次 keypad 都可以切换一次模式。



六、实验总结

1.KPU 是可以处理图像，让其产生不一样的视觉效果。

2.KPU 与 LCD 的数据格式是不一样的，需要转化以下才可以正常显示。

附：API

对应的头文件 kpu.h

kpu_task_init

描述

初始化 kpu 任务句柄，该函数具体实现在 model compiler 生成的 gencode_output.c 中。

函数定义

```
kpu_task_t* kpu_task_init(kpu_task_t* task)
```

参数

参数名称	描述	输入输出
task	KPU 任务句柄	输入

返回值

KPU 任务句柄。

kpu_run

描述

启动 KPU，进行 AI 运算。

函数原型

```
int kpu_run(kpu_task_t* v_task, dmac_channel_number_t dma_ch, const void *src,
void* dest, plic_irq_callback_t callback)
```

参数

参数名称	描述	输入输出
task	KPU 任务句柄	输入
dma_ch	DMA 通道	输入
src	输入图像数据	输入
dest	运算输出结果	输出
callback	运算完成回调函数	输入

返回值

返回值	描述
0	成功
非 0	KPU 忙，失败

kpu_get_output_buf

描述

获取 KPU 输出结果的缓存。

函数原型

```
uint8_t *kpu_get_output_buf(kpu_task_t* task)
```

参数

参数名称	描述	输入输出
task	KPU 任务句柄	输入

返回值

KPU 输出结果的缓存的指针。

kpu_release_output_buf

描述

释放 KPU 输出结果缓存。

函数原型

```
void kpu_release_output_buf(uint8_t *output_buf)
```

参数

参数名称	描述	输入输出
output_buf	KPU 输出结果缓存	输入

返回值

无

kpu_start

描述

启动 KPU，进行 AI 运算。

函数原型

```
int kpu_start(kpu_task_t *task)
```

参数

参数名称	描述	输入输出
task	KPU 任务句柄	输入

返回值

返回值	描述
0	成功
非 0	KPU 忙，失败

kpu_single_task_init

描述

初始化 kpu 任务句柄。

函数原型

```
int kpu_single_task_init(kpu_task_t *task)
```

参数

参数名称	描述	输入输出
task	KPU 任务句柄	输入

返回值

返回值	描述
0	成功
非 0	失败

kpu_single_task_deinit

描述

注销 kpu 任务。

函数原型

```
int kpu_single_task_deinit(kpu_task_t *task)
```

参数

参数名称	描述	输入输出
task	KPU 任务句柄	输入

返回值

返回值	描述
0	成功
非 0	失败

kpu_model_load_from_buffer

描述

解析 kmodel 并初始化 kpu 句柄。

函数原型

```
int kpu_model_load_from_buffer(kpu_task_t *task, uint8_t *buffer,
kpu_model_layer_metadata_t **meta);
```

参数

参数名称	描述	输入输出
task	KPU 任务句柄	输入
buffer	kmodel 数据	输入
meta	内部测试数据，用户设置为 NULL	输出

返回值

返回值	描述
0	成功
非 0	失败

kpu_load_kmodel

描述

加载 kmodel，需要与 nncase 配合使用。

函数原型

```
int kpu_load_kmodel(kpu_model_context_t *ctx, const uint8_t *buffer)
```

参数

参数名称	描述	输入输出
ctx	KPU 任务句柄	输入
buffer	kmodel 数据	输入

返回值

返回值	描述
0	成功
非 0	失败

kpu_model_free

描述

释放 kpu 资源。

函数原型

```
void kpu_model_free(kpu_model_context_t *ctx)
```

参数

参数名称	描述	输入输出
ctx	KPU 任务句柄	输入

返回值

无。

kpu_get_output

描述

获取 KPU 最终处理的结果。

函数原型

```
int kpu_get_output(kpu_model_context_t *ctx, uint32_t index, uint8_t **data,
size_t *size)
```

参数

参数名称	描述	输入输出
ctx	KPU 任务句柄	输入
index	结果的索引值，如 kmodel 有关	输入
data	结果	输入
size	大小（字节）	输入

返回值

返回值	描述
0	成功
非 0	失败

kpu_run_kmodel

描述

运行 kmodel。

函数原型

```
int kpu_run_kmodel(kpu_model_context_t *ctx, const uint8_t *src,
dmac_channel_number_t dma_ch, kpu_done_callback_t done_callback, void
*userdata)
```

参数

参数名称	描述	输入输出
ctx	KPU 任务句柄	输入
src	源数据	输入
dma_ch	DMA 通道	输入
done_callback	完成后回调函数	输入
userdata	回调的参数	输入

返回值

返回值	描述
0	成功
非 0	失败

举例

```
/* 通过 MC 生成 kpu_task_gencode_output_init，设置源数据为 g_ai_buf，使用 DMA5，kpu
完成后调用 ai_done 函数 */
kpu_task_t task;
volatile uint8_t g_ai_done_flag;
static int ai_done(void *ctx)
{
    g_ai_done_flag = 1;
    return 0;
}

/* 初始化 kpu */
kpu_task_gencode_output_init(&task); /* MC 生成的函数 */
task.src = g_ai_buf;
task.dma_ch = 5;
task.callback = ai_done;
kpu_single_task_init(&task);

/* 启动 kpu */
kpu_start(&task);
```

数据类型

相关数据类型、数据结构定义如下：

- kpu_task_t：kpu 任务结构体。

kpu_task_t

描述

kpu 任务结构体。

定义

```
typedef struct
{
```

```
    kpu_layer_argument_t *layers;
    kpu_layer_argument_t *remain_layers;
    plic_irq_callback_t callback;
    void *ctx;
    uint64_t *src;
    uint64_t *dst;
    uint32_t src_length;
    uint32_t dst_length;
    uint32_t layers_length;
    uint32_t remain_layers_length;
    dmac_channel_number_t dma_ch;
    uint32_t eight_bit_mode;
    float output_scale;
    float output_bias;
    float input_scale;
    float input_bias;
} kpu_task_t;
```

成员

成员名称	描述
layers	KPU 参数指针(MC 初始化，用户不必关心)
remain_layers	KPU 参数指针（运算过程中使用，用户不必关心）
callback	运算完成回调函数（需要用户设置）
ctx	回调函数的参数（非空需要用户设置）
src	运算源数据（需要用户设置）
dst	运算结果输出指针（KPU 初始化赋值，用户不必关心）
src_length	源数据长度(MC 初始化，用户不必关心)
dst_length	运算结果长度(MC 初始化，用户不必关心)
layers_length	层数(MC 初始化，用户不必关心)
remain_layers_length	剩余层数（运算过程中使用，用户不必关心）
dma_ch	使用的 DMA 通道号（需要用户设置）
eight_bit_mode	是否是 8 比特模式(MC 初始化，用户不必关心)
output_scale	输出 scale 值(MC 初始化，用户不必关心)
output_bias	输出 bias 值(MC 初始化，用户不必关心)
input_scale	输入 scale 值(MC 初始化，用户不必关心)
input_bias	输入 bias 值(MC 初始化，用户不必关心)