4.3 快速傅立叶变换加速器

一、实验目的

本节课主要学习 K210 芯片中快速傅立叶变换加速器的功能。

二、实验准备

1.实验元件

K210 芯片中的快速傅立叶变换加速器

2.元件特性

K210 内置快速傅立叶变换加速器 FFT Accelerater。

FFT 加速器是用硬件的方式来实现 FFT 的基 2 时分运算。

- 支持多种运算长度，即支持 64 点、128 点、256 点以及 512 点运算
- 支持两种运算模式，即 FFT 以及 IFFT 运算
- 支持可配的输入数据位宽，即支持 32 位及 64 位输入
- 支持可配的输入数据排列方式，即支持虚部、实部交替，纯实部以及实部、虚部分离三种数据排列方式
- 支持 DMA 传输

4.SDK 中对应 API 功能

对应的头文件 aes.h

为用户提供以下接口：

- fft_complex_uint16_dma：FFT 运算。

三、实验原理

目前该模块可以支持 64 点、128 点、256 点以及 512 点的 FFT 以及 IFFT。在 FFT 内部有两块大小为 512*32bit 的 SRAM，在配置完成后 FFT 会向 DMA 发送 TX 请求，将 DMA 送来的送据放到其中的一块 SRAM 中去，直到满足当前 FFT 运算所需要的数据量并开始 FFT 运算，蝶形运算单元从包含有有效数据的 SRAM 中读出数据，运算结束后将数据写到另外一块 SRAM 中去，下次蝶形运算再从刚写入的 SRAM 中读出数据，运算结束后并写入另外一块 SRAM，如此反复交替进行直到完成整个 FFT 运算。

四、实验过程

1. 首先通过三角函数取得一组复数。

```
int32_t i;
float tempf1[3];
fft_data_t *output_data;
fft_data_t *input_data;
uint16_t bit1_num = get_bit1_num(FFT_FORWARD_SHIFT);
complex_hard_t data_hard[FFT_N] = {0};
complex data_soft[FFT_N] = {0};
/* 取得一组复数 */
for (i = 0; i < FFT_N; i++)
{
    tempf1[0] = 0.3 * cosf(2 * PI * i / FFT_N + PI / 3) * 256;
    tempf1[1] = 0.1 * cosf(16 * 2 * PI * i / FFT_N - PI / 9) * 256;
    tempf1[2] = 0.5 * cosf((19 * 2 * PI * i / FFT_N) + PI / 6) * 256;
    data_hard[i].real = (int16_t)(tempf1[0] + tempf1[1] + tempf1[2] + 10);
    data_hard[i].imag = (int16_t)0;
    data_soft[i].real = data_hard[i].real;
    data_soft[i].imag = data_hard[i].imag;
}
```

2. 将取得的复数转化成快速傅立叶变换的数据结构，作为输入的数据（待计算）。

```
/* 复数转化成傅里叶数据结构RIRI */
for (int i = 0; i < FFT_N / 2; ++i)
{
    input_data = (fft_data_t *)&buffer_input[i];
    input_data->R1 = data_hard[2 * i].real;
    input_data->I1 = data_hard[2 * i].imag;
    input_data->R2 = data_hard[2 * i + 1].real;
    input_data->I2 = data_hard[2 * i + 1].imag;
}
```

3. 分别使用硬件和软件进行快速傅立叶变换运行，并记录运行的消耗时间。

```
/* 硬件处理FFT数据，并记录时间 */
cycle[FFT_HARD][FFT_DIR_FORWARD] = read_cycle();
fft_complex_uint16_dma(DMAC_CHANNEL0, DMAC_CHANNEL1, FFT_FORWARD_SHIFT, FFT_DIR_FORWARD,
cycle[FFT_HARD][FFT_DIR_FORWARD] = read_cycle() - cycle[FFT_HARD][FFT_DIR_FORWARD];

/* 软件处理FFT数据，并记录时间 */
cycle[FFT_SOFT][FFT_DIR_FORWARD] = read_cycle();
fft_soft(data_soft, FFT_N);
cycle[FFT_SOFT][FFT_DIR_FORWARD] = read_cycle() - cycle[FFT_SOFT][FFT_DIR_FORWARD];
```

4. 对输出的数据进行取模。

```
/* 复数取模 */
for (i = 0; i < FFT_N; i++)
{
    hard_power[i] = sqrt(data_hard[i].real * data_hard[i].real + data_hard[i].imag * data
    soft_power[i] = sqrt(data_soft[i].real * data_soft[i].real + data_soft[i].imag * data
}
```

5. 打印复数的实部和虚部的数据，以及模和相位等信息。

```
/* 打印软件和硬件复数的实部和虚部 */
printf("\n[hard fft real][soft fft real][hard fft imag][soft fft imag]\n");
for (i = 0; i < FFT_N / 2; i++)
    printf("%3d:%7d %7d %7d %7d\n",
           i, data_hard[i].real, (int32_t)data_soft[i].real, data_hard[i].imag, (int32_t)data_soft[i].imag);

printf("\nhard power  soft power:\n");
printf("%3d : %f  %f\n", 0, hard_power[0] / 2 / FFT_N * (1 << bit1_num), soft_power[0] / 2 / FFT_N);
for (i = 1; i < FFT_N / 2; i++)
    printf("%3d : %f  %f\n", i, hard_power[i] / FFT_N * (1 << bit1_num), soft_power[i] / FFT_N);

/* 打印相位 */
printf("\nhard phase  soft phase:\n");
for (i = 0; i < FFT_N / 2; i++)
{
    hard_angel[i] = atan2(data_hard[i].imag, data_hard[i].real);
    soft_angel[i] = atan2(data_soft[i].imag, data_soft[i].real);
    printf("%3d : %f  %f\n", i, hard_angel[i] * 180 / PI, soft_angel[i] * 180 / PI);
}
```

6. 接下来是快速傅立叶变换逆运算，把刚刚计算出来的数据放回去逆运算。

```
/* 快速傅里叶变换逆运算 */
for (int i = 0; i < FFT_N / 2; ++i)
{
    input_data = (fft_data_t *)&buffer_input[i];
    input_data->R1 = data_hard[2 * i].real;
    input_data->I1 = data_hard[2 * i].imag;
    input_data->R2 = data_hard[2 * i + 1].real;
    input_data->I2 = data_hard[2 * i + 1].imag;
}

/* 硬件和软件快速傅里叶变换运算 */
cycle[FFT_HARD][FFT_DIR_BACKWARD] = read_cycle();
fft_complex_uint16_dma(DMAC_CHANNEL0, DMAC_CHANNEL1, FFT_BACKWARD_SHIFT, FFT_DIR_BACKWARD, buffer_input, FFT_N, buffer_output);
cycle[FFT_HARD][FFT_DIR_BACKWARD] = read_cycle() - cycle[FFT_HARD][FFT_DIR_BACKWARD];
cycle[FFT_SOFT][FFT_DIR_BACKWARD] = read_cycle();
ifft_soft(data_soft, FFT_N);
cycle[FFT_SOFT][FFT_DIR_BACKWARD] = read_cycle() - cycle[FFT_SOFT][FFT_DIR_BACKWARD];
```

7. 打印快速傅立叶变换逆运算的输出数据。

```
for (i = 0; i < FFT_N / 2; i++)
{
    output_data = (fft_data_t*)&buffer_output[i];
    data_hard[2 * i].imag = output_data->I1 ;
    data_hard[2 * i].real = output_data->R1 ;
    data_hard[2 * i + 1].imag = output_data->I2 ;
    data_hard[2 * i + 1].real = output_data->R2 ;
}

printf("\n[hard ifft real][soft ifft real][hard ifft imag][soft ifft imag]\n");
for (i = 0; i < FFT_N / 2; i++)
    printf("%3d:%7d  %7d %7d %7d\n",
           i, data_hard[i].real, (int32_t)data_soft[i].real, data_hard[i].imag, (int32_t)data_soft[i].imag);
```

8. 打印硬件和软件计算傅立叶变换和逆运算的时间作为对比。

```
printf("[hard fft test] [%d bytes] forward time = %ld us, backward time = %ld us\n",
        FFT_N,
        cycle[FFT_HARD][FFT_DIR_FORWARD]/(sysctl_clock_get_freq(SYSCTL_CLOCK_CPU)/1000000),
        cycle[FFT_HARD][FFT_DIR_BACKWARD]/(sysctl_clock_get_freq(SYSCTL_CLOCK_CPU)/1000000));

printf("[soft fft test] [%d bytes] forward time = %ld us, backward time = %ld us\n",
        FFT_N,
        cycle[FFT_SOFT][FFT_DIR_FORWARD]/(sysctl_clock_get_freq(SYSCTL_CLOCK_CPU)/1000000),
        cycle[FFT_SOFT][FFT_DIR_BACKWARD]/(sysctl_clock_get_freq(SYSCTL_CLOCK_CPU)/1000000));
while (1)
    ;
```

9.编译调试，烧录运行

把本课程资料中的 fft 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=fft -G "MinGW Makefiles"

make

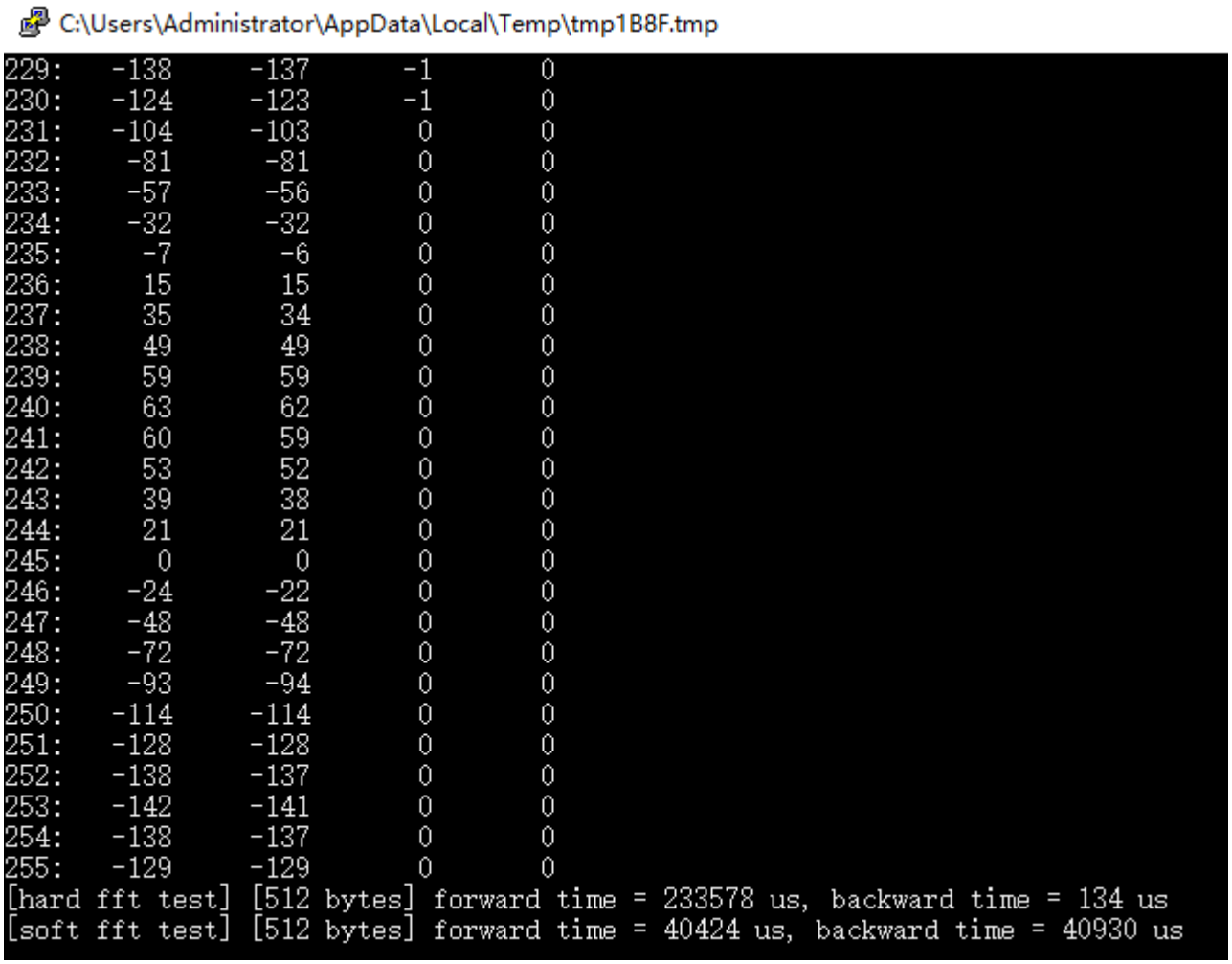
```
Generating .bin file ...
[100%] Built target fft
PS C:\K210\SDK\kendryte-standalone-sdk-develop\build>
```

编译完成后，在 build 文件夹下会生成 fft.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，并且打印傅立叶变换和逆运算的数据，以及两种运算过程消耗的时间。



六、实验总结

1.单独使用 CPU 也可以实现 FFT 计算。

2.软件和硬件的 FFT 计算的时间会有比较大的差异。

附：API

对应的头文件 fft.h

fft_complex_uint16_dma

描述

FFT 运算。

函数原型

```
void fft_complex_uint16_dma(dmac_channel_number_t dma_send_channel_num,
dmac_channel_number_t dma_receive_channel_num, uint16_t shift,
fft_direction_t direction, const uint64_t *input, size_t point_num, uint64_t
*output);
```

参数

参数名称	描述	输入输出
dma_send_channel_num	发送数据使用的 DMA 通道号	输入
dma_receive_channel_num	接收数据使用的 DMA 通道号	输入
shift	FFT 模块 16 位寄存器导致数据溢出(-32768~32767)，FFT 变换有 9 层，shift 决定哪一层需要移位操作(如 0x1ff 表示 9 层都做移位操作；0x03 表示第第一层与第二层做移位操作)，防止溢出。如果移位了，则变换后的幅值不是正常 FFT 变换的幅值，对应关系可以参考 fft_test 测试 demo 程序。包含了求解频率点、相位、幅值的示例	输入
direction	FFT 正变换或是逆变换	输入
input	输入的数据序列，格式为 RIRI..,实部与虚部的精度都为 16bit	输入
point_num	待运算的数据点数，只能为 512/256/128/64	输入
output	运算后结果。格式为 RIRI..,实部与虚部的精度都为 16bit	输出

返回值

无。

举例

```
#define FFT_N              512U
#define FFT_FORWARD_SHIFT   0x0U
#define FFT_BACKWARD_SHIFT  0x1ffU
```

```
#define PI                3.14159265358979323846
complex_hard_t data_hard[FFT_N] = {0};
for (i = 0; i < FFT_N; i++)
{
    tempf1[0] = 0.3 * cosf(2 * PI * i / FFT_N + PI / 3) * 256;
    tempf1[1] = 0.1 * cosf(16 * 2 * PI * i / FFT_N - PI / 9) * 256;
    tempf1[2] = 0.5 * cosf((19 * 2 * PI * i / FFT_N) + PI / 6) * 256;
    data_hard[i].real = (int16_t)(tempf1[0] + tempf1[1] + tempf1[2] + 10);
    data_hard[i].imag = (int16_t)0;
}
for (int i = 0; i < FFT_N / 2; ++i)
{
    input_data = (fft_data_t *)&buffer_input[i];
    input_data->R1 = data_hard[2 * i].real;
    input_data->I1 = data_hard[2 * i].imag;
    input_data->R2 = data_hard[2 * i + 1].real;
    input_data->I2 = data_hard[2 * i + 1].imag;
}
fft_complex_uint16_dma(DMAC_CHANNEL0, DMAC_CHANNEL1, FFT_FORWARD_SHIFT,
FFT_DIR_FORWARD, buffer_input, FFT_N, buffer_output);
for (i = 0; i < FFT_N / 2; i++)
{
    output_data = (fft_data_t*)&buffer_output[i];
    data_hard[2 * i].imag = output_data->I1 ;
    data_hard[2 * i].real = output_data->R1 ;
    data_hard[2 * i + 1].imag = output_data->I2 ;
    data_hard[2 * i + 1].real = output_data->R2 ;
}
for (int i = 0; i < FFT_N / 2; ++i)
{
    input_data = (fft_data_t *)&buffer_input[i];
    input_data->R1 = data_hard[2 * i].real;
    input_data->I1 = data_hard[2 * i].imag;
    input_data->R2 = data_hard[2 * i + 1].real;
    input_data->I2 = data_hard[2 * i + 1].imag;
}
fft_complex_uint16_dma(DMAC_CHANNEL0, DMAC_CHANNEL1, FFT_BACKWARD_SHIFT,
FFT_DIR_BACKWARD, buffer_input, FFT_N, buffer_output);
for (i = 0; i < FFT_N / 2; i++)
{
    output_data = (fft_data_t*)&buffer_output[i];
    data_hard[2 * i].imag = output_data->I1 ;
    data_hard[2 * i].real = output_data->R1 ;
    data_hard[2 * i + 1].imag = output_data->I2 ;
```

```
    data_hard[2 * i + 1].real = output_data->R2 ;
}
```

数据类型

相关数据类型、数据结构定义如下：

- fft_data_t：FFT 运算传入的数据格式。
- fft_direction_t：FFT 变换模式。

fft_data_t

描述

FFT 运算传入的数据格式。

定义

```
typedef struct tag_fft_data
{
    int16_t I1;
    int16_t R1;
    int16_t I2;
    int16_t R2;
} fft_data_t;
```

成员

成员名称	描述
I1	第一个数据的虚部
R1	第一个数据的实部
I2	第二个数据的虚部
R2	第二个数据的实部

fft_direction_t

描述

FFT 变换模式

定义

```
typedef enum _fft_direction
```

```
{
    FFT_DIR_BACKWARD,
    FFT_DIR_FORWARD,
    FFT_DIR_MAX,
} fft_direction_t;
```

成员

成员名称	描述
FFT_DIR_BACKWARD	FFT 逆变换
FFT_DIR_FORWARD	FFT 正变换